(2)

AD-A200 756

IDA MEMORANDUM REPORT M-362

Ada/SQL BINDING SPECIFICATIONS

Bill R. Brykczynski
Fred Friedman

June 1988

Prepared for
WIS Joint Program Office

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

IDA Log No. HQ 87-032695

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

AD-A200 756

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Memorandum Report M-362	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION OUSDA, DIMO
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311
8a NAME OF FUNDING/SPONSORING ORGANIZATION WIS Joint Program Office 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) WIS JPMO/XPT Washington, D.C. 20330-6600	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-W5-206 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Ada/SQL Binding Specifications (U)	
12 PERSONAL AUTHOR(S) Bill R. Brykczynski, Fred Friedman	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT (Year, Month, Day) 1988 June 15 PAGE COUNT 314
16 SUPPLEMENTARY NOTATION	

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada programming language; structured query language (SQL); database management system (DBMS); interface; binding; database definition language (DDL); database manipulation language (DML).

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of IDA Memorandum Report M-362 is to document the current version of the Ada/SQL specifications. These specifications have been previously reported in IDA Paper P-1944, *Preliminary Version: Ada/SQL: A Standard, Portable DBMS Interface*. This Memorandum Report is intended to provide a formal reference for the Ada/SQL language specification. M-362 can be used by those persons interested in understanding the underlying concepts of Ada/SQL and by those persons implementing Ada/SQL systems.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Mr. Bill R. Brykczynski	22b TELEPHONE (Include area code) (703) 824-5515	22c OFFICE SYMBOL IDA/CSED

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

IDA MEMORANDUM REPORT M-362

ADa/SQL BINDING SPECIFICATIONS

Bill R. Brykcyznski
Fred Friedman

June 1988

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-W5-206

Executive Summary

1. Introduction

The purpose of IDA Memorandum Report M-362, *Ada/SQL Binding Specifications*, is to document the current version of the Ada/SQL specifications. These specifications have been previously reported in IDA Paper P-1944, *Preliminary Version: Ada/SQL: A Standard, Portable DBMS Interface* [IDA 86]. Prototype software implementing this interface has been reported in [IDA 86], and IDA Memorandum Report M-361, *Example Level 1 System Software* [IDA 88a], and also IDA Memorandum Report M-459, *An Oracle - Ada/SQL Implementation* [IDA 88b].

1.1 Scope

IDA Memorandum Report M-362 is intended to provide a formal reference for the Ada/SQL language specification. M-362 can be used by those persons interested in understanding the underlying concepts of Ada/SQL and by those persons implementing Ada/SQL systems. The section numbers of the main body of the specification map closely to the section numbers (clauses) of the ANSI SQL specification.

1.2 Background

Ada/SQL is a binding between the Ada programming language and the database programming language SQL. Ada/SQL provides a Database Definition Language (DDL) and a Database Manipulation Language (DML) capability, both expressed by compilable Ada statements. Objects defined using the Ada/SQL DDL can be expressed using the Ada data typing facilities. These objects are then subject to the strong type checking of the Ada compiler. The Ada/SQL DML is defined as Ada subprogram calls which are modeled after the SQL DML. The Ada/SQL DML subprograms provide the link between the Ada program and the underlying Database Management System (DBMS). [IDA 86] provides an overview of the concepts behind the design of the Ada/SQL language.

This new Ada/SQL specification improves upon the [IDA 86] version in the following ways:

- It provides precise and formal definition of the language, in the style of the ANSI SQL standard, thereby making the relation of Ada/SQL to ANSI SQL completely clear.

- It provides actual Ada declarations that demonstrate precisely how Ada/SQL statements could be compiled by a validated Ada compiler, although it does not require an implementation to actually provide those declarations (similar declarations are actually used in several implementations; Ada/SQL programs can also be preprocessed, changing the form of the Ada/SQL statements).

- It provides notes describing how the Ada declarations were designed, the rationale behind certain features of the language, how Ada/SQL relates to ANSI SQL, etc.

- It is based on actual implementation experience, and so includes those features that have been found to be readily implementable and fully compatible with ANSI SQL, while deferring other features for later enhancements.

- It provides a comprehensive index, to aid the reader in tracing the web of references to various constructs, that of necessity becomes labyrinthine in a standard of this sort.

1.3 References

[IDA 86] Brykczynski, Bill and Fred Friedman. 1986. *Preliminary version: Ada/SQL: A standard, portable Ada-DBMS interface.* Alexandria, VA: Institute for Defense Analyses. IDA Paper P-1944.

[IDA 88a] Brykczynski, Bill, Fred Friedman, and Kerry Hilliard. 1988. *Example Level 1 System Software.* Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-423.

[IDA 88b] Brykczynski, Bill, Fred Friedman, and Kerry Hilliard. 1988. *An Oracle-Ada/SQL implementation.* Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-459.

[SQL86] ANSI X3.135-1986, Database Language SQL. New York, NY: American National Standards Institute, Inc., 1986.

Preface

The purpose of IDA Memorandum Report M-362, *Ada/SQL Binding Specifications*, is to present a revised set of specifications previously reported in IDA Paper P-1944, *Preliminary Version: Ada/SQL: A Standard, Portable DBMS Interface*. This revision to the previous specifications is a result of experience gained in additional prototyping, suggestions from users, and a desire to model the specification after the ANSI SQL [SQL86] document style.

The importance of this document is based on partial fulfillment of Task Order T-W5-206, WIS Application Software Study. These specifications will be used by the WIS program in specifying an Ada interface to database management systems.

This document was funded in part by the Software Technology for Adaptable, Reliable Systems (STARS) Joint Program Office. Special thanks is given to the STARS JPO for their guidance and support.

CONTENTS

Database Language - Ada/SQL

1. Scope and field of application

This standard specifies the syntax and semantics of two database languages:

1) A schema definition language (Ada/SQL-DDL), for declaring the structures and integrity constraints of an Ada/SQL database.

2) A data manipulation language (Ada/SQL-DML), for declaring the executable statements of a specific database application program.

This standard defines the logical data structures and basic operations for an Ada/SQL database. It provides functional capabilities for designing, accessing, maintaining, controlling, and protecting the database.

This standard provides a vehicle for portability of database definitions and application programs between conforming implementations.

Ada/SQL is designed as an extension to ANSI SQL, adding Ada's type declaration and checking capabilities to SQL, and expressing data manipulation operations in standard, compilable Ada. It is designed such that it can be implemented as a front-end to a database management system implementing Level 2 of the ANSI standard SQL.

NOTE: Additional Ada/SQL language is planned for later addenda to this standard. Major topics under consideration for such addenda include referential integrity, enhanced transaction management, specification of certain implementor-defined rules, enhanced character handling facilities, and support for national character sets.

This standard applies to implementations that exist in an environment that may include application programming languages, end-user query languages, report generator systems, data dictionary systems, program library systems, and distributed communication systems, as well as various tools for database design, data administration, and performance optimization.

2. References

This standard is intended for use with the following American National Standards. When these standards are superceded by revisions approved by the American National Standards Institute, the revisions shall apply.

ANSI/MIL-STD-1815A (1983), Military Standard Ada Programming Language.

ANSI X3.135-1986, Database Language - SQL.

3. Overview

3.1 Organization

The organization of this standard is as follows:

1) 3.2, "Notation" and 3.3, "Conventions" define the notations and conventions used in this standard.

2) 3.4, "Conformance" defines conformance criteria.

3) 3.5, "Example database" presents the definition of the database used in the examples.

4) Clause 4, "Concepts" defines terms and presents concepts used in the definition of Ada/SQL.

5) Clause 5, "Common elements" defines language elements that occur in several parts of the Ada/SQL language.

6) Clause 6, "Schema definition language" defines the Ada/SQL facilities for specifying a database.

7) Clause 7, "Program environment" defines the way Ada/SQL statements are included in Ada programs.

8) Clause 8, "Data manipulation language" defines the data manipulation statements of Ada/SQL.

3.2 Notation

The syntactic notation used in this standard is BNF ("Backus Normal Form", or "Backus-Naur Form"), with the following extensions:

1) Square brackets ([]) indicate optional elements.

2) Ellipses (...) indicate elements that may be repeated one or more times.

3) Braces ({}) group sequences of elements.

In the BNF syntax, a production symbol <A> is defined to "contain" a production symbol <B> if <B> occurs someplace in the expansion of <A>. If <A> contains <B>, then <B> is "contained in" <A>. If <A> contains <B>, then <A> is the "containing" <A> production symbol for <B>.

A production symbol <A> is defined to "immediately contain" a production symbol <B> if <B> occurs someplace in the expansion of <A>, and there is no other instance of production symbol <A>

in the expansion that also contains that particular instance of <B>. If <A> immediately contains <B>, then <B> is "immediately contained in" <A>. If <A> immediately contains <B>, then <A> is the "immediately containing" <A> production symbol for <B>.

References to Ada production symbols use the Ada notation, which can readily be distinguished from Ada/SQL notation. Ada/SQL production symbol names are enclosed in angle brackets, with words separated by spaces. Ada production symbol names are not enclosed in angle brackets, and words are separated by underscores.

3.3 Conventions

Syntactic elements of this standard are specified in terms of:

1) *Function*: A short statement of the purpose of the element.

2) *Format*: A BNF definition of the syntax of the element.

3) *Effective Ada Declarations*: Ada specifications for constructs that could be used to enable Ada compilation of the element.

4) *Example*: Examples of the use of the element, in complete Ada/SQL statements.

5) *Syntax Rules*: Additional syntactic constraints not expressed in BNF that the element shall satisfy.

6) *General Rules*: A sequential specification of the run-time effect of the element.

7) *Notes*: Comments on design decisions made in defining Ada/SQL, the rationale behind the effective Ada declarations, and how Ada/SQL relates to ANSI SQL. The Notes also include indications of divergence of Release 1 implementations from this standard. Release 1 implementations chronologically preceded this standard, did not implement all features, and implemented some features slightly differently.

NOTE: The effective Ada declarations are presented in a style designed to be demonstrative, rather than rigorous. If the declarations are assumed to all be contained within a single declarative region, then the order of presentation in this document is not the order required for visibility of dependent declarations. Naming conflicts that may arise due to capricious choice of table and/or type names are also not addressed.

In the Syntax Rules, the term "shall" defines conditions that are required to be true of syntactically conforming Ada/SQL language. The treatment of Ada/SQL language that does not conform to the Formats or the Syntax Rules shall be of one of the two following implementor-defined forms:

1) A language processor reading Ada/SQL source language shall flag the error, or

2) The exception SYNTAX_ERROR shall be raised when a program attempts to execute a nonconforming statement.

In the General Rules, the term "shall" defines conditions that are to be tested at run-time during the execution of Ada/SQL statements. If all such conditions are true, then the statement executes successfully and no exceptions are raised. If any such condition is false, then the statement does not execute successfully, the statement execution has no effect on the database, and an exception is raised as noted in the rule, or the program executing the statement is erroneous.

NOTE: There are certain rules that must be obeyed by Ada/SQL programs, but, due to the potential difficulty of detecting their violation, for which this standard does not require Ada/SQL systems to

detect violation. Any program causing one of these rules to be violated is considered erroneous, and the effect of erroneous execution is unpredictable. An implementation detecting violations of such rules may raise the appropriate exception, as noted in the rules.

NOTE: The ANSI SQL concept of SQLCODE is not used in Ada/SQL; status returns are instead handled by Ada exceptions. Enhancements to this standard are planned to provide additional status information. Release 1 implementations have exception names differing from those specified in this standard. (The exception names used here are based on planned ANSI SQL enhancements; specification of these enhancements was not yet available when the Release 1 implementations were designed.) The correspondence of exception names is as follows:

This standard name	Release 1 implementation name
NO_DATA	NOT_FOUND_ERROR
DATA_EXCEPTION, CONSTRAINT_ERROR	CONSTRAINT_ERROR, NULL_ERROR *
INVALID_CURSOR_STATE	INVALID_CURSOR_STATE_ERROR
CARDINALITY_VIOLATION	UNIQUE_ERROR

*Release 1 implementations raise one or the other of these two exceptions for conditions where this standard requires either DATA_EXCEPTION or CONSTRAINT_ERROR to be raised. In general, DATA_EXCEPTION is used to indicate data values out of range or otherwise in error, and is chosen to be consistent with new emerging ANSI SQL standards. However, there are instances where implementations dependent on the effective Ada declarations cannot raise DATA_EXCEPTION, but must instead depend on Ada semantics to raise CONSTRAINT_ERROR. Hence, this standard requires CONSTRAINT_ERROR, rather than DATA_EXCEPTION, to be raised in those cases, which involve the assignment of database values to program variables.

A conforming implementation is not required to perform the exact sequence of actions defined in the General Rules, but shall achieve the same effect on the database and the executing program as that sequence. The term "effectively" is used in the General Rules to emphasize actions whose effect might be achieved in other ways by an implementation.

The term "persistent object" is used to characterize objects such as <schema>s that are created and destroyed by implementor-defined mechanisms.

NOTE: Enhancements to this standard are planned to define the mechanisms for creating and destroying persistent objects.

In this standard, clauses begin on a new page, and in clause 5, "Common elements" through clause 8, "Data manipulation language" subclauses begin a new page. The resulting white space is not significant.

3.4 Conformance

This standard specifies conforming Ada/SQL language and conforming Ada/SQL implementations. Conforming Ada/SQL language shall abide by the BNF Format and associated Syntax Rules. A conforming Ada/SQL implementation shall process standard conforming Ada/SQL language according to

the General Rules.

A conforming implementation may provide additional facilities or options not specified by this standard, but only insofar as they do not invalidate any of the BNF Formats, General Rules, or Syntax Rules of this standard. A conforming implementation must report the appropriate errors when processing non-conforming Ada/SQL language, and must process conforming Ada/SQL in a conforming manner.

A conforming implementation need not actually produce the effective Ada declarations shown in this standard, although definition of the corresponding Ada bodies would enable execution of Ada/SQL statements; such an implementation is called a *runtime* system. For example, a conforming implementation might read Ada/SQL programs and produce modified source code, replacing Ada/SQL statements with embedded ANSI SQL statements or calls upon procedures of an ANSI module. Such an implementation is called a *preprocessed* system.

3.5 Example Database

The database used for the examples is designed to be demonstrative of Ada/SQL constructs, not of database programming practices. It is therefore very concise and simple. The examples assume that all declarations have been made directly visible. The example database is as follows:

```
with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package EXAMPLE_AUTHORIZATION is

  function EXAMPLE is new AUTHORIZATION_IDENTIFIER;

end EXAMPLE_AUTHORIZATION;

package EXAMPLE_TYPES is

  package ADA_SQL is

    type EMPLOYEE_NAME is new STRING ( 1 .. 30 );

    type BOSS_NAME is new EMPLOYEE_NAME;

    type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

    type HOURLY_WAGE_FOR_COMPUTATIONS is new EMPLOYEE_SALARY;

    subtype HOURLY_WAGE is HOURLY_WAGE_FOR_COMPUTATIONS range 0.00 .. 48.08;

  end ADA_SQL;

end EXAMPLE_TYPES;
```

```
with SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION, EXAMPLE_TYPES;
 use SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION;

package EXAMPLE_SDL is

  package ADA_SQL is

    use EXAMPLE_TYPES.ADA_SQL;

    SCHEMA_AUTHORIZATION : IDENTIFIER := EXAMPLE;

    subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;

    type EMPLOYEE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type NEW_EMPLOYEE_FILE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type ONE_EMPLOYEE_TABLE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type MANAGERS is
      record
        NAME : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
      end record;

  end ADA_SQL;

end EXAMPLE_SDL;
```

For convenience, the declarations of program variables used with each example are shown with the example. If the examples are to be taken as part of an actual conforming Ada/SQL program, the declarations should, of course, be considered to be placed in one or more <global variable package>s or <local variable package>s, and then made directly visible from the examples.

4. Concepts

4.1 Sets

A set is an unordered collection of distinct objects.

A multi-set is an unordered collection of objects that are not necessarily distinct.

A sequence is an ordered collection of objects that are not necessarily distinct.

The cardinality of a collection is the number of objects in that collection. Unless specified otherwise, any collection may be empty.

4.2 Data types

A *type* is a set of representable values. A value belongs to a type if it is within the set. The logical representation of a value is a <literal>. The physical representation of a value is implementor-defined, but must provide all the logical properties required by this specification, as well as those required by the Ada standard when values are stored in Ada program variables.

The set of possible values within a type can be further subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to satisfy a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint; a value is said to belong to a subtype of a given type if it belongs to the type and satisfies the constraint.

The declaration of a type specifies a name to be used to refer to the type and may also specify a constraint. Values of a type are represented in a format that may permit values not satisfying the constraint (if any); the set of values relevant to the representation is called the *base type*. Ada program representation may be different from database representation, so the set of values in a base type may differ depending on whether they are program values or database values. Where it is necessary to make the distinction, *base type* refers to the database base type, while *Ada base type* specifically refers to the program base type.

For <type declaration>s not syntactically permitting a constraint, the name specified denotes the base type. For <type declaration>s permitting a constraint, the base type is anonymous (not named), and the name specified denotes the subtype which includes all the values of the base type if no constraint is actually specified, or, if a constraint is given, all the values of the base type satisfying the constraint. This subtype denoted by the name is called the *first-named subtype*. The term *data type* is used to refer to the base type or first-named subtype, as appropriate, denoted by a name declared in a <type declaration>.

Just as a <type declaration> declares a name as denoting a data type, so does a <subtype declaration> declare a name as denoting a subtype. However, a base type is also a subtype of itself; such a subtype corresponds to a condition that imposes no restriction. Consequently, the name declared in a <type declaration> also denotes the appropriate subtype.

A value is primitive, in that it has no logical subdivision within this standard. (Although individual

characters of character strings can be accessed in Ada, they may not be individually accessed in Ada/SQL statements within Ada programs.) A value is a null value or a nonnull value. (Null values may be stored in a database, but not in Ada program variables. When referring to values in an Ada program, <indicator variable>s and <indicator value>s are used to flag whether or not the values are null. A null Ada character string is not the same as the null Ada/SQL character string value.)

A null value is an implementor-defined data type dependent special value that is distinct from all nonnull values of that data type.

A nonnull value is of one of four classes of data type: integer, floating point, character string, or enumeration. Only values of the same data type are comparable, however, provision is made for converting between closely related data types.

Typing is applied to program values, which result from evaluating program variables, <literal>s, and <named number>s; program variables used to receive values retrieved from a database via <target specification>s; database columns; and database values, which result from the execution and evaluation of database operations.

A value assigned to a program variable or a database column must belong to the subtype of the variable or column. An exception is raised if assignment of a value out of range is attempted to a program variable. Constraint checking on assignment to database columns is more difficult; an implementation may raise an exception if possible. In any case, programs assigning values out of range to database columns are erroneous.

4.2.1 Character strings

A character string consists of a sequence of characters of the ASCII character set. Although they cannot be individually accessed from Ada/SQL, each character in a character string has an integer index, with the index of each character being one less than the index of the following character (if any). A character string subtype is characterized by: (1) the subtype of index, (2) whether it is constrained or unconstrained, and, if constrained, the lower and upper bounds of the index (same as those of the index subtype), and (3) the subtype of the characters. All characters (if any) within a character string must belong to the subtype defined for the characters of that character string data type.

A character string program value or program object has a lower index bound, L, and an upper index bound, U.

If the character string is of an unconstrained subtype and L is less than or equal to U, then both L and U must lie within the range of the index subtype. The length of the character string is U-L+1. If the subtype of the character string is unconstrained and L is greater than U, then the length of the character string is zero (a string of no characters, or a null Ada character string).

If the character string is of a constrained subtype, then L and U must be equal to the lower and upper bounds, respectively, of the index subtype declared for the character string subtype. All character strings of the same constrained subtype are of the same length, U-L+1.

A character string database value is not characterized by its index bounds, but only by its length, the number of characters it contains. All character strings derived from the same database column are considered to have the same length, whether the column is of a constrained or an unconstrained subtype. For a constrained subtype, this length is the length declared for the subtype. For an unconstrained subtype, the length is the maximum length possible, i.e., u-l+1, where l and u are the lower and upper

bounds, respectively, of the index subtype.

To account for the differing treatment of character strings between a program and a database, the following effects applying when transferring values between program and database:

1) When a character string taken from the database is stored in a program variable, the first character of the string to be stored is stored in the first character of the variable, then successive characters of the string are stored in successive characters of the variable.

2) When a character string taken from the database is stored in a program variable, a <last variable> is set to indicate the index of the last character stored. The character string stored is equal to that taken from the database, except that one or more trailing blanks may be removed from the database value. The treatment of trailing blanks is implementation dependent, and any program whose effect depends on the treatment of trailing blanks is erroneous. If a character string value contains more characters than does the program variable into which it is to be stored, an exception is raised. Note that storing a particular character string value into a program variable may raise an exception in one implementation but not in another, depending on the way trailing blanks are handled.

   NOTE: The index of the last character stored will be the appropriate index value in the character string program variable, except in the case where: (1) the <out variable> into which the character string value is to be stored is expressed as <type mark> (<variable name>) (an Ada type conversion), (2) the data type denoted by the <type mark> is a constrained character string data type, and (3) the index bounds of the constrained character string data type are not the same as the index bounds of the program variable denoted by the <variable name> (both index bounds must encompass the same number of characters, however). In this case, the index value that is stored is that appropriate for the constrained character string data type denoted by the <type mark>. For example, consider a constrained character string data type with index bounds of 1..10, and a character string program variable with index bounds of 11..20. If five characters are stored in the variable, the <last variable> will be set to 5; the appropriate index value in the program variable is 15.

3) When a character string program value is stored in a database column, it is padded with trailing blanks as necessary to the appropriate length. If the character string program value is longer than the database column will permit, the program causing the store is erroneous.

Two character strings are comparable if and only if they are of the same data type. A value of any character string data type can be converted to any other character string data type. A character string is identical to another character string if and only if it is equal to that character string in accordance with the comparison rules specified in 5.11, "<comparison predicate>".

4.2.2 Numbers

A number is either an integer value or a floating point value. Two numbers are comparable if and only if they are of the same data type. A value of any integer data type can be converted to any other integer data type. A value of any integer or floating point data type can be converted to any floating point data type. An Ada program value of any floating point data type can be converted to any integer data data type – this latter conversion is not permitted for database values.

An integer subtype is characterized by the lower and upper bounds of the range of values it contains. All integers between the lower and upper bounds, inclusive, belong to the subtype. A range with the lower bound greater than the upper bound is a null range; no integers belong to the subtype.

A floating point subtype is characterized by its accuracy and by the lower and upper bounds of the range of values it contains. The accuracy is a positive integer that specifies the minimum number of significant decimal digits. All real numbers between the lower and upper bounds, inclusive, belong to the subtype, and are represented in both the program and the database to at least the accuracy specified. (The representations used in the program and the database may be different.) A range with the lower bound greater than the upper bound is a null range; no real numbers belong to the subtype.

Assignment of an integer value to an integer program variable or database column is exact.

Whenever an integer or floating point value is assigned to a floating point program variable or database column, an approximation of its value is represented in the data type of the target, retaining at least the accuracy of the target.

4.2.3 Enumeration types

An enumeration data type is characterized by its enumeration literal values. Both characters and identifiers may be used as enumeration literals. An enumeration data type with only character enumeration literals is said to be a character data type. Two enumeration values are comparable if and only if they are of the same data type. A value of an enumeration data type A may be converted to a value of another enumeration data type B if and only if there exists a data type C such that (1) C is not a derived type, (2) either A is the same as C or A is derived from C, and (3) either B is the same as C or B is derived from C. C is called the ultimate parent type of both A and B.

4.2.4 Derived types

A derived data type belongs to one of the four classes of data type: character string, integer, floating point, or enumeration. It inherits the characteristics of the data type from which it is derived, which is called the parent data type. The way in which data types are derived from each other affects the convertibility of enumeration values.

4.3 Columns

A column is a multi-set of values that may vary over time. All values of the same column are of the same data type and are values in the same table. A value of a column is the smallest unit of data that can be selected from a table and the smallest unit of data that can be updated.

A column has a description and an ordinal position within a table. The description of a column includes its data type and an indication of whether the column is constrained to contain only nonnull values. The description of a character string column includes its length attribute. The description of a floating point numeric column includes the accuracy and range of its numbers. The description of an integer numeric column includes the range of its numbers. The description of an enumeration column includes the enumeration literals. Note that column and program variable descriptions, for the purpose of determining comparability and convertibility, depend on the data type, not the subtype. When values are stored in a column or a program variable, subtype constraints may be checked to determine the legality of the operation.

A named column is a column of a named table or a column that inherits the description of a named column. The description of a named column includes its name.

4.4 Tables

A table is a multi-set of rows. A row is a nonempty sequence of values. Every row of the same table has the same cardinality and contains a value of every column of that table. The i-th value in every row of a table is a value of the i-th column of that table. The row is the smallest unit of data that can be inserted into a table and deleted from a table.

The degree of a table is the number of columns of that table. At any time, the degree of a table is the same as the cardinality of each of its rows and the cardinality of a table is the same as the cardinality of each of its columns.

A table has a description. The description includes a description of each of its columns.

A base table is a named table defined by a <table definition>. The description of a base table includes its name.

A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a <query specification>. The values of a derived table are those of the underlying tables when it is derived.

A viewed table is a named derived table defined by a <view definition>. The description of a viewed table includes its name.

A table is either updatable or read-only. The operations of insert, update, and delete are permitted for updatable tables and are not permitted for read-only tables.

A grouped table is a set of groups derived during the evaluation of a <group by clause>. A group is a multi-set of rows in which all values of the grouping column(s) are equal. A grouped table may be considered as a collection of tables. Set functions may operate on the individual tables within the grouped table.

A grouped view is a viewed table derived from a grouped table.

4.5 Integrity constraints

Integrity constraints define the valid states of the database by constraining the values in the base tables. Constraints may be defined to prevent two rows in a table from having the same values in a specified column (_NOT_NULL_UNIQUE subtype suffix) or columns (<unique constraint definition>) or to prevent a column from containing a null value (_NOT_NULL subtype suffix).

Integrity constraints are effectively checked after execution of each <SQL statement>. If the base table associated with an integrity constraint does not satisfy that integrity constraint, then the <SQL statement> has no effect and the appropriate exception is raised.

Data types and subtypes may also constrain the ranges of values that may be stored within a column. Requiring subtype constraint checking may significantly impair the performance of Ada/SQL implementations dependent on database management systems not performing such checking, so subtype constraint checking is not required by this standard. An implementation performing subtype constraint

checking may raise the DATA_EXCEPTION exception for violation of the constraint. The execution of a program that would cause a subtype constraint to be violated is erroneous. Subtype constraints are always checked on assignment of database values to Ada program variables.

4.6 Schemas

A <schema> is a persistent object specified by the schema definition language. It consists of a <schema authorization clause> and all <table definition>s, <unique constraint definition>s, <view definition>s, and <privilege definition>s known to the system for a specified <authorization identifier> in an environment. The concept of environment is implementor-defined.

The tables, views, and privileges defined by a <schema> are considered to be "owned by" or to have been "created by" the <authorization identifier> specified for that <schema>.

NOTE: An implementation may provide facilities (such as DROP TABLE, DROP VIEW, ALTER TABLE, and REVOKE) that allow the definitions of the tables, views, and privileges for a given <authorization identifier> to be created, destroyed, and modified incrementally over time. This standard, however, only addresses the <schema>s that represent the definitions known to the system at a given time.

4.7 The database

The database is the collection of all data defined by the <schema>s in an environment. The concept of environment is implementor-defined.

4.8 Program Environment

An Ada/SQL application program is a segment of executable code, possibly consisting of multiple compilation units. Also part of the program environment to be referenced by application program compilation units are: (1) the schema definition language, including definitions of database data types, tables, and columns, (2) <global variable package>s and <local variable package>s, defining all the variables that will be used for communication between the program and the database, as well as <correlation name>s referenced in <SQL statement>s, and (3) the Ada/SQL predefined environment, containing declarations that facilitate writing portable programs.

4.9 Ada-SQL interface

Ada/SQL provides Ada programs with the ability to manipulate data controlled by a database management system, and also to exchange data with that database management system. The exchange of data is accomplished between values in Ada program variables and values in database columns. Treatment of values in Ada program variables is according to Ada semantics. Treatment of values in database columns is, minimally, according to SQL semantics. Ada/SQL systems may also implement extensions to SQL that allow values in database columns to be treated more in accordance with Ada semantics (e.g., subtype checking).

4.10 Status indicators

4.10.1 Execution status

No special indication is provided for an Ada/SQL statement that executes successfully. An exception is raised when an attempted Ada/SQL statement does not execute successfully.

NOTE: Extensions to Ada/SQL are planned that will provide more detailed status reporting.

4.10.2 Indicators

Ada program variables assume or supply the values exchanged with the database. Ada program variables must, when evaluated, have a defined value. There is a special database null value, however, that is different from all other values of a particular data type. This value is not represented in an Ada program variable of that type; indicators are instead used to indicate whether or not a value is null. An indicator has a value of enumeration type INDICATOR_VARIABLE, with literals NULL_VALUE and NOT_NULL. When supplying a value to the database, a program <indicator value> determines whether or not the supplied value is null. When retrieving a database value into a program variable, an associated <indicator variable> is set to indicate whether or not the retrieved value is null.

4.11 Standard programming language

This standard specifies the actions of <SQL statement>s when those <SQL statement>s are invoked by programs that conform to the standard Ada programming language. The term "standard Ada program" refers to programs that meet the conformance criteria of the Ada standard listed in clause 2, "References".

4.12 Cursors

A cursor is specified by a <declare cursor>.

For each <declare cursor> in a program, a cursor is effectively created by the execution of that <declare cursor> and destroyed when the program defining it terminates or when another cursor is created with the same <cursor name>.

NOTE: This is different from the ANSI SQL definition of cursor creation and destruction, but the differences do not have any operational impact on database effects.

A cursor is in either the open state or the closed state. The initial state of a cursor is the closed state. A cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement>, a <commit statement>, or a <rollback statement>.

A cursor in the open state designates a table, an ordering of the rows of that table, and a position relative to that ordering. If the <declare cursor> does not specify an <order by clause>, then the rows of the table have an implementor-defined order. This order is subject to the reproducibility requirement within

a transaction (see 4.16, "Transactions"), but it may change between transactions. Any program whose effect depends on the ordering of rows in such a table is erroneous.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row even though the table is empty.

A <fetch statement> advances the position of an open cursor to the next row of the cursor's ordering and retrieves the values of the columns of that row. An <update statement: positioned> updates the current row of the cursor. A <delete statement: positioned> deletes the current row of the cursor.

If a cursor is before a row and a new row is inserted at that position, then the effect, if any, on the position of the cursor is implementor-defined. Any program whose effect depends on that position is erroneous.

If a cursor is on a row or before a row and that row is deleted, then the cursor is positioned before the row that is immediately after the position of the deleted row. If such a row does not exist, then the position of the cursor is after the last row.

If an error occurs during the execution of an <SQL statement> that identifies an open cursor, then the effect, if any, on the position or state of that cursor is implementor-defined. Any program whose effect depends on that position is erroneous.

A working table is a table resulting from the opening of a cursor. Whether opening a cursor results in creation of a working base table or a working viewed table is implementor-defined. Any program whose effect depends on this distinction is erroneous.

Each row of a working viewed table is derived only when the cursor is positioned on that row.

A working base table is created when the cursor is opened and destroyed when the cursor is closed.

4.13 Statements

An <SQL statement> specifies a database operation or a cursor operation, or declares a cursor. A <select statement> fetches values from a table. An <insert statement> inserts rows into a table. An <update statement: searched> or <update statement: positioned> updates the values in rows of a table. A <delete statement: searched> or <delete statement: positioned> deletes rows of a table.

4.14 Embedded syntax

An Ada/SQL program is an application program that consists of Ada programming language text and Ada/SQL text. The Ada programming language text shall conform to the requirements of the standard Ada programming language. The Ada/SQL text, consisting of one or more <SQL statement>s as defined in this standard, shall also conform to the requirements of the standard Ada programming language, when taken with the program environment and the effective Ada declarations. This allows database applications to be expressed in pure Ada, with the <SQL statement>s embedded directly in an application program.

4.15 Privileges

A *privilege* authorizes a given category of <action> to be performed on a specified table or view by a specified <authorization identifier>. The <action>s that can be specified are INSERT, DELETE, SELEC, and UPDATE.

An <authorization identifier> is specified for each <schema package> declaring database tables or views, and is implicit for each execution of an Ada/SQL program. The association of an <authorization identifier> with a particular execution of an Ada/SQL program is implementor-defined. The effect of a program, particularly with respect to privileges, may vary from implementation to implementation, depending on how the <authorization identifier> is associated with a program execution.

All tables, views, and privileges declared in <schema package>s with the same <authorization identifier> are part of the same <schema>, which is also considered to have that <authorization identifier>. The <authorization identifier> of a <schema> is the "owner" of all tables and views defined in that <schema>.

Tables and views are designated by <table name>s. A <table name> consists of an <authorization identifier> and a <table identifier>. The <authorization identifier> identifies the <schema> in which the table or view designated by the <table name> was defined. Tables and views defined in different <schema>s can have the same <table identifier>.

If a reference to a <table name> within a <schema> does not explicitly contain an <authorization identifier>, then the <authorization identifier> of the containing <schema> is specified by default. If a reference to a <table name> within an <SQL statement> does not explicitly contain an <authorization identifier>, then exactly one of the <schema>s referenced from the compilation unit containing the <SQL statement> shall declare a table with the identifier used in the <table name>, and the <authorization identifier> of that <schema> is specified by default.

The <authorization identifier> of a <schema> has all privileges on the tables and views defined in that <schema>.

A <schema> with a given <authorization identifier> may contain <privilege definition>s that grant privileges to other <authorization identifier>s. The granted privileges may apply to tables and views defined in the current <schema>, or they may be privileges that were granted to the given <authorization identifier> by other <schema>s. The WITH_GRANT_OPTION clause of a <privilege definition> specifies whether the recipient of a privilege may grant it to others.

The <authorization identifier> implicitly associated with each execution of an Ada/SQL program shall have the privileges specified for each <SQL statement> executed by the program; otherwise, an exception is raised for an attempt to execute an <SQL statement> for which the required privileges are lacking.

4.16 Transactions

A transaction is a sequence of operations, including database operations, that is atomic with respect to recovery and concurrency. A transaction is initiated when a program executes an <SQL statement> and no transaction is currently active. All tasks within the same program participate in the same transaction.

A transaction is terminated by a <commit statement>, a <rollback statement>, an <exit database statement>, or program termination. A transaction terminated by an <exit database statement> or program termination is terminated as if a <rollback statement> had been executed. If a transaction is terminated by a <commit statement>, then all changes made to the database by that transaction are made accessible to all concurrent transactions. If a transaction is terminated by a <rollback statement>, then all changes made to the database by that transaction are canceled. Committed changes cannot be canceled. Changes made to the database by a transaction can be perceived by that transaction, but until that transaction terminates with a <commit statement> they cannot be perceived by other transactions.

The execution of concurrent transactions is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing transactions that produces the same effect as some serial execution of those same transactions. A serial execution is one in which each transaction executes to completion before the next transaction begins.

The execution of an <SQL statement> within a transaction has no effect on the database other than the effect stated in the General Rules for that <SQL statement>. Together with serializable execution, this implies that all read operations are reproducible within a transaction, except for changes explicitly made by the transaction itself.

5. Common elements

5.1 <character>

Function

Define the terminal symbols of the language and the elements of strings.

Format

<character> ::=
<digit> | <letter> | <special character>

<digit> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::=
<upper case letter> | <lower case letter>

<upper case letter> ::=
A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | X

<lower case letter> ::=
a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | x

<special character> ::=
" # & ' ( ) * + , - . / : ; < = > _ | ! $ % ? @ [ ] ^ ` { } ~
| <space>

Effective Ada Declarations

None.

Example

Not applicable.

Syntax Rules

1) With respect to the format of <special character>: The list of <special character>s on the first line is to be read as if a BNF "|" were to appear between each pair of <special character>s; the "|"s are omitted for clarity. The "|" <special character> on the first line is to be read as itself, not a BNF metasymbol. The "|" character on the second line is the BNF "|" metasymbol.

General Rules

None.

Notes

1) Ada/SQL <character>s conform to ANSI SQL <character>s. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1-SR2	SR1	2

2) Ada/SQL <special character>s are explicitly specified, to be the same as Ada's. The <special character>s specified are in full compliance with ANSI SQL SR1 and SR2.

5.2 <literal>

Function

Specify a nonnull value.

Format

```
<literal> ::=
    <character string literal>
  | <numeric literal>
  | <enumeration literal>

<character string literal> ::=
    " <character representation> ... "

<character representation> ::=
    <nonquote character>
  | <quote representation>

<nonquote character> ::= <character>

<quote representation> ::= ""

<numeric literal> ::=
    <integer literal>
  | <floating point literal>

<integer literal> ::=
    <integer> [ <exponent> ]
  | <base> # <based integer> # [ <exponent> ]

<floating point literal> ::=
    <integer> . <integer> [ <exponent> ]
  | <base> # <based integer> . <based integer> # [ <exponent> ]

<integer> ::=
    <digit> [ { [ <underscore> ] <digit> } ... ]

<exponent> ::=
    E [ + ] <integer>
  | E - <integer>

<base> ::= <integer>

<based integer> ::=
    <extended digit> [ { [ <underscore> ] <extended digit> } ... ]

<extended digit> ::=
    <digit>
```

| <letter>

<enumeration literal> ::=
  <library package name> . ADA_SQL . <simple enumeration literal>
| <library package name> . <simple enumeration literal>
| <simple enumeration literal>

<simple enumeration literal> ::=
  <program identifier>
| <character literal>

<character literal> ::=
  ' <character> '

Effective Ada Declarations

None.

Example

```
"Message of the day:"
""                    -- null character string; not legal in Ada/SQL
" "     "A"     """" -- three string literals of length 1
"Characters such as $, %, and ) are allowed in string literals"

12          0           1E6         123_456     -- integer literals
12.0        0.0         0.456       3.14159_26 -- floating point literals
1.34E-12    1.0E+6 -- floating point literals with exponent

2#1111_1111#      16#FF#            016#0FF# -- integer literals of value 256
16#E#E1           2#1110_0000#               -- integer literals of value 224
16#F.FF#E+2       2#1.1111_1111_111#E11      -- floating point literals of
                                             --  value 4095.0

PHONE_PACKAGE.ADA_SQL.'0' -- user-defined enumeration literals (assuming
'0'                       --  definitions shown below)
STANDARD.TRUE -- predefined enumeration literals
TRUE
```

The declaration of PHONE_PACKAGE might be:

```
package PHONE_PACKAGE is
  package ADA_SQL is
    type PHONE_NUMBER_CHARACTER is new CHARACTER range '0' .. '9';
  end ADA_SQL;
end PHONE_PACKAGE;
```

Syntax Rules

1) A <nonquote character> shall not contain the double quote mark character (").

2) The data type of a <character string literal> is some character string data type which, as a consequence of other Syntax Rules, is determinable solely from the context in which the <character string literal> appears. If a <character string literal> is contained in an <Ada type qualification>, then its subtype is that denoted by the <type mark> of the <Ada type qualification>; otherwise, its subtype is the same as its data type. Ada visibility rules apply to the characters within a <character string literal>. The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote representation> in a <character string literal> represents a single quotation mark character in both the value and the length of the <character string literal>. Spaces are significant <character representation>s within <character string literal>s; they are shown in the BNF only to delimit notational elements.

3) Spaces shall not appear within the text of a <numeric literal>; they are shown in the BNF only to delimit notational elements.

4) The letter E of the <exponent>, if any, can be written either in lower case or in upper case, with the same meaning. An <exponent> contained in an <integer literal> shall not contain a minus sign.

5) A <base> must be at least two and at most sixteen. The only <letter>s allowed as <extended digit>s are the letters A through F, representing the digits ten through fifteen. A <letter> in a <based integer> can be written either in lower case or in upper case, with the same meaning. The value of each <extended digit> within a <numeric literal> must be less than the <base> of that <numeric literal>.

6) Arithmetic on <integer literal>s (and integer <named number>s) is performed without regard to the constraints of any particular data type, as if they were of a *universal integer* data type. When interpretation as a value of a specific integer data type is required by the context, an <integer literal> or an otherwise untyped expression of <integer literal>s and/or integer <named number>s is taken to be of that data type.

7) Arithmetic on <floating point literal>s (and floating point <named number>s) is performed without regard to the constraints of any particular data type, as if they were of a *universal floating point* data type. When interpretation as a value of a specific floating point data type is required by the context, a <floating point literal> or an otherwise untyped expression of <floating point literal>s and/or floating point <named number>s is taken to be of that data type. (Universal floating point expressions may also contain <integer literal>s and integer <named number>s in certain contexts; see 5.6, <value specification>, and 5.9, <value expression>.)

8) A <character literal> shall consist of exactly one <character> between two apostrophes; the extra spaces are shown in the BNF only to delimit notational elements.

9) Case:

   a) If an <enumeration literal> contains the <library package name> STANDARD, then it shall be of the form

      <library package name> . <simple enumeration literal>

      and the same <simple enumeration literal> shall be declared within at least one effective

<enumeration type> within the STANDARD Ada/SQL predefined environment. Let the set of effective <enumeration type>s containing the same <simple enumeration literal> be called the *relevant <enumeration type>s*.

b) If an <enumeration literal> contains a <library package name> denoting a library package that is part of the Ada/SQL predefined environment, then it shall be of the form

<library package name> . <simple enumeration literal>

and the same <simple enumeration literal> shall be declared within at least one effective <enumeration type> within that library package. Let the set of effective <enumeration type>s containing the same <simple enumeration literal> be called the *relevant <enumeration type>s*.

c) If an <enumeration literal> contains a <library package name> denoting a library package that is not part of the Ada/SQL predefined environment, then it shall be of the form

<library package name> . ADA_SQL . <simple enumeration literal>

and the same <simple enumeration literal> shall be declared within at least one <enumeration type> within the ADA_SQL nested package of that library package. Let the set of <enumeration type>s containing the same <simple enumeration literal> be called the *relevant <enumeration type>s*.

d) If an <enumeration literal> is of the form

<simple enumeration literal>

then the same <simple enumeration literal> shall be declared within at least one of the following:

an effective <enumeration type> within the STANDARD Ada/SQL predefined environment, and/or

one or more <enumeration type>s contained within any innermost package denoted by a <package name> contained in a <use clause> that applies to the <schema package body> or <Ada/SQL DML unit> containing the <enumeration literal>.

Furthermore, the <simple enumeration literal> shall not be the same as the name of any data type, subtype, table, <named number>, or variable declared within the STANDARD Ada/SQL predefined environment or within any innermost package denoted by a <package name> contained in a <use clause> that applies to the <schema package body> or <Ada/SQL DML unit> containing the <enumeration literal>. Also, the <simple enumeration literal> shall be directly visible by Ada rules.

Let the set of <enumeration type>s declaring the same <simple enumeration literal> be called the *relevant <enumeration type>s*.

10) The data type of an <enumeration literal> is that declared by one of the relevant <enumeration type>s. Where there is more than one relevant <enumeration type>, the one selected is that required by other Syntax Rules expressing Ada/SQL's strong typing.

General Rules

1) The value of a <character string literal> is the sequence of <character>s that it contains, with each <quote representation> representing a single quotation mark <character>.

2) Case:

   a) If a <character string literal> is of a constrained character string subtype, then its lower and upper bounds are given by those of the index subtype of the constrained character string subtype. The length of the <character string literal> shall be the same as the upper bound minus the lower bound, plus one; otherwise, the CONSTRAINT_ERROR exception is raised.

   b) If a <character string literal> is of an unconstrained character string subtype, then its lower bound is given by the lower bound of the index subtype of the unconstrained character string subtype, and its upper bound is equal to the lower bound, plus the length of the <character string literal>, minus one. The upper bound shall belong to the index subtype; otherwise, the CONSTRAINT_ERROR exception is raised.

3) An <underscore> character inserted between adjacent digits of an <integer> or a <based integer> does not affect its value.

4) Case:

   a) If a <numeric literal> does not contain a <base>, then its value is that expressed by the conventional decimal notation (that is, the base is implicitly ten). An <exponent> indicates the power of ten by which the value of the <numeric literal> without the <exponent> is to be multiplied to obtain the value of the <numeric literal> with the <exponent>.

   b) If a <numeric literal> does contain a <base>, then its value is that expressed by the conventional based notation, except that the <base> and the <exponent>, if any, are in decimal notation. An <exponent> indicates the power of the <base> by which the value of the <numeric literal> without the <exponent> is to be multiplied to obtain the value of the <numeric literal> with the <exponent>.

5) The value of an <integer literal> or an untyped expression of <integer literal>s and/or integer <named number>s, when taken to be of a specific integer data type, shall be exact. The value shall belong to the data type; otherwise, the CONSTRAINT_ERROR exception is raised.

6) The value of a <floating point literal> or an untyped expression containing <floating point literal>s and/or floating point <named number>s, when taken to be of a specific floating point data type, shall be within the accuracy of that data type. The value shall belong to the data type; otherwise, the CONSTRAINT_ERROR exception is raised.

7) The value of an <enumeration literal> is that which it denotes within its data type.

Notes

1) Ada/SQL <literal>s conform to ANSI SQL <literal>s. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	2
SR2	SR2	3
SR3	SR3-SR5	4
SR4-SR5	SR6-SR7	5
—	SR8-SR10	6
GR1	GR1	-
—	GR2	7
GR2-GR3	GR3-GR4	8
—	GR5-GR6	9
—	GR7	-

2) Ada/SQL uses quotation mark characters for string brackets, as does Ada; ANSI SQL uses apostrophes (which they call single quotation marks). Note that Ada/SQL syntax does not permit <character string literal>s containing no <character representation>s, even though such null string literals are permitted in Ada. This is because ANSI SQL also requires at least one <character representation> in a <character string literal>.

3) Ada/SQL SR2 expresses one aspect of Ada/SQL's strong typing, consistent with Ada's rules for string literals.

4) Ada/SQL <numeric literal> syntax complies with that of Ada, differing from the ANSI SQL syntax in the following ways:

   a) ANSI SQL allows a leading plus or minus sign; Ada/SQL does not. The value of an Ada/SQL <numeric literal> can, of course, be negated by putting a minus sign in front of it; the minus sign is taken to be a monadic operator instead of part of the <numeric literal>. Ada/SQL is also more restrictive than ANSI SQL on where monadic operators may be used (see 5.6, <value specification>, and 5.9, <value expression>).

   b) Ada/SQL allows the E for the <exponent> to be written in either upper or lower case; ANSI SQL requires upper case.

   c) An Ada/SQL <integer literal> may contain an <exponent>, subject to certain restrictions. The ANSI SQL analog, <exact numeric literal>s, may not contain <exponent>s.

d) An Ada/SQL <floating point literal> containing a radix point (period) must have at least one digit on each side of the point; the <mantissa> of the ANSI SQL analog, <approximate numeric literal>, may begin or end with a decimal point.

e) Ada/SQL allows based notation; ANSI SQL only supports base ten.

f) Ada/SQL permits <underscore>s to be used as digit separators; ANSI SQL does not allow digit separators.

5) Ada/SQL SR6 and SR7 express aspects of Ada/SQL's strong typing, consistent with Ada's implicit type conversions from types *universal_integer* and *universal_real*.

6) Ada/SQL SR9 basically restates Ada name syntax and visibility considerations for <enumeration literal>s. It is still necessary to add that a <simple enumeration literal> used without qualification shall be directly visible by Ada rules, because <use clause>s may refer to <non Ada/SQL package name>s, and certain Ada use_clauses do not fall under the purview of Ada/SQL syntax. These elements may affect the visibility of <simple enumeration literal>s.

7) The determination of lower and upper bounds for Ada/SQL <character string literal>s is that prescribed by Ada for string literals.

8) Ada/SQL <numeric literal>s are interpreted according to Ada rules, which go beyond those of ANSI SQL in allowing <underscore>s as digit separators and based notation.

9) Raising CONSTRAINT_ERROR corresponds to the Ada semantics on implicit type conversions.

5.3 <token>

Function

Specify lexical units.

Format

<token> ::=
    <nondelimiter token> | <delimiter token>

<nondelimiter token> ::=
    <identifier>
  | <key word>
  | <numeric literal>

<identifier> ::=
    <letter> [{ [<underscore>] <letter or digit> } ...]

<underscore> ::= _

<letter or digit> ::=
    <letter> | <digit>

<database identifier> ::= <identifier>

<program identifier> ::= <identifier>

<key word> ::=
    <Ada reserved word>
  | <SQL key word>
  | <Ada/SQL statement name>
  | <Ada/SQL reserved word>

<Ada reserved word> ::=

abort	declare	for	new	raise	task
abs	delay	function	not	range	terminate
accept	delta		null	record	then
access	digits	generic		rem	type
all	do	goto	of	renames	
and			or	return	use
array	else	if	others	reverse	
at	elsif	in	out		when
end	is		select	while	
begin	entry		package	separate	with
body	exception	limited	pragma	subtype	
exit	loop	private		xor	
case			procedure		
constant		mod			

```
<SQL key word> ::=
      ALL           | END           | LANGUAGE      | SCHEMA
    | AND           | ESCAPE        | LIKE          | SECTION
    | ANY           | EXEC                          | SELECT
    | AS            | EXISTS        | MAX           | SET
    | ASC                           | MIN           | SMALLINT
    | AUTHORIZATION | FETCH         | MODULE        | SOME
    | AVG           | FLOAT                         | SQL
                    | FOR           | NOT           | SQLCODE
    | BEGIN         | FORTRAN       | NULL          | SQLERROR
    | BETWEEN       | FOUND         | NUMERIC       | SUM
    | BY            | FROM
                                    | OF            | TABLE
    | CHAR          | GO            | ON            | TO
    | CHARACTER     | GOTO          | OPEN
    | CHECK         | GRANT         | OPTION        | UNION
    | CLOSE         | GROUP         | OR            | UNIQUE
    | COBOL                         | ORDER         | UPDATE
    | COMMIT        | HAVING                        | USER
    | CONTINUE                      | PASCAL
    | COUNT         | IN            | PLI           | VALUES
    | CREATE        | INDICATOR     | PRECISION     | VIEW
    | CURRENT       | INSERT        | PRIVILEGES
    | CURSOR        | INT           | PROCEDURE     | WHENEVER
                    | INTEGER       | PUBLIC        | WHERE
    | DEC           | INTO                          | WITH
    | DECIMAL       | IS            | REAL          | WORK
    | DECLARE                       | ROLLBACK
    | DELETE
    | DESC
    | DISTINCT
    | DOUBLE

<Ada/SQL statement name> ::=
      CLOSE           | EXIT_DATABASE | INSERT          | SELEC
    | COMMIT_WORK                    | INSERT_INTO     | SELEC_ALL
    | CONSTRAINTS     | FETCH         | INTO            | SELEC_DISTINCT
    | CREATE_VIEW                                      | SELECT_ALL
                      | GRANT         | OPEN            | SELECT_DISTINCT
    | DECLAR                         | OPEN_DATABASE
    | DELETE                                           | UPDATE
    | DELETE_FROM                    | ROLLBACK_WORK

<Ada/SQL reserved word> ::=
      ALL_PRIVILEGES | DESC          | MAX             | SOME
    | ALLL                           | MAX_ALL         | SUM
    | ANY            | ENABLED       | MAX_DISTINCT    | SUM_ALL
    | ASC            | EQ            | MIN             | SUM_DISTINCT
    | AVG            | EXISTS        | MIN_ALL
    | AVG_ALL                         | MIN_DISTINCT    | UNION
    | AVG_DISTINCT   | IDENTIFIER                      | UNION_ALL
                     | INDICATOR     | NE              | UNIQUE
```

```
    | BETWEEN          | IS_IN           | NOT_IN          | USER
                       | IS_NOT_NULL     | NOT_NULL
    | CONVERT_TO       | IS_NULL         | NULL_VALUE      | VALUES
    | COUNT
    | COUNT_ALL        | LIKE            | PUBLIC
    | COUNT_DISTINCT
```

<delimiter token> ::=
  <character string literal>
 | <character literal>
 | , | (|) | < | > | . | : | * | + | -
 | / | >= | <= | & | ' | ; | => | .. | := | <>

<separator> ::=
  { <comment> | <space> | <newline> | <format effector> } ...

<comment> ::=
  -- [<comment character> ...] <newline>

<comment character> ::=
  <character> | <horizontal tabulation>

<horizontal tabulation> ::=
  ASCII horizontal tabulation character

<newline> ::=
  implementor-defined end-of-line indicator

<space> ::=
  ASCII space character

<format effector> ::=
  <horizontal tabulation>
 | *ASCII vertical tabulation character*
 | *ASCII carriage return character*
 | *ASCII line feed character*
 | *ASCII form feed character*

Effective Ada Declarations

see sections relevant to various key/reserved words

Example

```
NAME          name9          LAST_NAME      FirstName      -- <identifier>s
TOO_LONG_FOR_A_DATABASE_IDENTIFIER

-- <comment>s:
---------------- the first two hyphens start the comment
COMMIT_WORK;--spaces are not necessary around "--" to start comment
```

Syntax Rules

1) A <token>, other than a <character string literal> or a space <character literal>, shall not include a <space>.

2) Any <token> may be followed by a <separator>. A <nondelimiter token> shall be followed by a <delimiter token> or a <separator>. If the syntax does not allow a <nondelimiter token> to be followed by a <delimiter token>, then that <nondelimiter token> shall be followed by a <separator>.

3) A <space> within a <comment>, a <character string literal>, or a space <character literal> is not a <separator>. The spaces shown within the definition of <comment> are not required; they are shown in the BNF only to delimit notational elements. <horizontal tabulation> within a <comment> is not a <separator>.

4) <newline> is an implementor-defined end-of-line indicator. If, for a given implementation, the end of a line is signified by one or more characters, then these characters shall be <format effector>s other than <horizontal tabulation>. In any case, a sequence of one or more <format effector>s other than <horizontal tabulation> shall cause at least one <newline>.

5) The single special characters shown for <delimiter token> are not <delimiter token>s when part of a two-character <delimiter token>, or contained within a <comment>, <character string literal>, <character literal>, or <numeric literal>. The two-character sequences shown for <delimiter token> are not <delimiter token>s when contained within a <comment> or <character string literal>.

6) All characters of an <identifier> are significant, including any <underscore> character inserted between a <letter> or <digit> and an adjacent <letter> or <digit>. <identifier>s differing only in the use of corresponding upper and lower case letters are considered as the same. No <space> is allowed within an identifier; spaces are shown in the BNF only to delimit notational components.

7) A <database identifier> shall not consist of more than 18 <character>s.

8) A <database identifier> shall not be identical to a <key word>; a <program identifier> shall not be identical to an <Ada reserved word>, an <Ada/SQL statement name>, or an <Ada/SQL reserved word>.

9) Let the term *potential homograph* be defined, in terms of Ada visibility rules, for a specific Ada identifier at a specific point in an Ada compilation unit, as follows: An identifier is a *potential homograph* if:

   a) A declaration of that identifier which does not allow overloading is directly visible, or

   b) A declaration of that identifier which does not allow overloading is hidden, or

   c) A declaration of that identifier which does not allow overloading is potentially visible, whether actually made directly visible or not.

10) At the point of its use, a <database identifier> shall not be a potential homograph.

11) At the point of its use, the appropriate declaration of a <program identifier>, as required by other Syntax Rules, shall be visible according to Ada visibility rules.

12) An <Ada/SQL statement name>, other than OPEN, CLOSE, or DELETE, shall not be used within an <Ada/SQL compilation unit>, except as explicitly prescribed by the syntax of this standard. Furthermore, at the point of its use, an <Ada/SQL statement name> shall not be a potential homograph.

13) At the point of its use, an <Ada/SQL reserved word> shall not be a potential homograph.

14) At the point of its use in a <correlation name declaration>, the identifier a_t_CORRELATION or t_CORRELATION, where "a" represents an <authorization identifier> and "t" represents a <table identifier>, shall not be a potential homograph.

General Rules

None.

Notes

1) Ada/SQL <token>s conform to ANSI SQL <token>s. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	2
SR2	SR2	-
—	SR3-SR5	3
—	SR6	4
SR3	SR7	5
SR4	SR8	6
—	SR9-SR10	7
—	SR11	8
—	SR12	9
—	SR13	10
—	SR14	11

2) ANSI SQL does not have <character literal>s.

3) Ada/SQL SR3, SR4, and SR5 correspond to Ada rules for tokens.

4) ANSI SQL only allows upper case within <identifier>s; Ada/SQL allows either upper or lower case, but ignores case in determining the meaning of an <identifier>.

5) ANSI SQL restricts all <identifier>s to no more than 18 characters. To maintain compatibility, Ada/SQL likewise restricts names of database objects, <database identifier>s, to 18 characters. No maximum length is placed on names of program objects, <program identifier>s, in accordance with Ada philosophy.

6) <database identifier>s and <program identifier>s must be distinct from Ada reserved word>s to comply with Ada syntax, and are required to be distinct from <Ada/SQL statement name>s and <Ada/SQL reserved word>s to facilitate parsing of Ada/SQL. <database identifier>s, which name database objects, are required to be different from <SQL key word>s to retain compatibility with ANSI SQL. <program identifier>s, which name program objects, may be identical with <SQL key word>s, providing they are also different from the <key word>s in the other categories.

   The restrictions of Ada/SQL SR8 would prohibit the use of any type COUNT, declared in package TEXT_IO and generic package DIRECT_IO, within Ada/SQL statements. At this time, neither TEXT_IO nor DIRECT_IO is part of the Ada/SQL predefined environment, so that the COUNT types are not available to Ada/SQL statements anyway. A later version of this standard may include TEXT_IO and/or DIRECT_IO in the Ada/SQL predefined environment, at which time some allowance would have to be made for COUNT.

7) In a runtime system, <database identifier>s are the names of functions which are made directly visible with a use clause. It would not be possible to make them directly visible, however, if they were potential homographs.

8) The "other Syntax Rules" referred to in Ada/SQL SR11 define how a <program identifier> must be declared. The purpose of SR11 is to ensure that those declarations will not be hidden or otherwise not directly visible due to the effects of other Ada declarations

9) The restriction on the use of <Ada/SQL statement name>s is placed by Ada/SQL SR12 to simplify the task of writing language processors for Ada/SQL. A language processor can ignore Ada constructs not part of the Ada/SQL language, merely scanning source text for each <Ada/SQL statement name>. Later versions of this standard may lift this restriction, which means that language processors would have to search more carefully for Ada/SQL statements. OPEN, CLOSE, and DELETE are allowed non-Ada/SQL uses because of their frequent programming usage, including in the predefined packages for Ada input-output. For these <key word>s, therefore, Ada/SQL language processors must already determine whether or not they are used as <Ada/SQL statement name>s.

   In a runtime system, <Ada/SQL statement name>s are the names of subprograms which are made directly visible with a use clause. It would not be possible to make them directly visible, however, if they were potential homographs. OPEN, CLOSE, and DELETE are declared as procedures in the predefined Ada input-output packages, hence, overloading is allowed for the names, and they are not potential homographs.

10) In a runtime system, <Ada/SQL reserved word>s are the names of functions which are made directly visible with a use clause. It would not be possible to make them directly visible, however, if they were potential homographs.

Note that package TEXT_IO and generic package DIRECT_IO declare types named COUNT. If TEXT_IO or a package instantiated from DIRECT_IO is named in an Ada use clause of an <Ada/SQL compilation unit>, then COUNT, which is also an <Ada/SQL reserved word> for use in COUNT ('*'), is a potential homograph. For this reason, Ada/SQL provides COUNT_ALL as a synonym which may be used in contexts where COUNT is prohibited.

Release 1 implementations do not support the COUNT_ALL synonym for COUNT.

11) In a runtime system, the a_t_CORRELATION or t_CORRELATION identifiers are package names which are made directly visible with a use clause. It would not be possible to make them directly visible, however, if they were potential homographs.

5.4 Names

Function

Specify names.

Format

<table name> ::= [<authorization identifier> .] <table identifier>

<authorization identifier> ::= <database identifier>

<table identifier> ::= <database identifier>

<column name> ::= <database identifier>

<correlation name> ::= <database identifier>

<cursor name> ::= <variable name>

<package name> ::=
    <unit simple name>
  | <unit simple name> . ADA_SQL
  | ADA_SQL
  | <non Ada/SQL package name>

<non Ada/SQL package name> ::= *a package name, the first identifier of which is a not a <library package name>*

<library package name> ::= <program identifier>

<non Ada/SQL library unit name> ::= *the name of an Ada library unit which satisfies the rules given in Syntax Rule 1 of 6.1.4*

<package identifier> ::= <program identifier>

<type mark> ::=
    <library package name> . ADA_SQL . <type identifier>
  | <library package name> . <type identifier>
  | <type identifier>

<type identifier> ::= <program identifier>

<program object name> ::=
    <variable name>
  | <named number name>

<variable name> :=
    <library package name> . <simple variable name>
  | ADA_SQL . <simple variable name>

| <simple variable name>

<simple variable name> ::= <program identifier>

<named number name> ::=
  <library package name> . ADA_SQL . <named number>
| <library package name> . <named number>
| <named number>

Effective Ada Declarations

In the DATABASE predefined package:

```
type USER_AUTHORIZATION_IDENTIFIER is new STANDARD.STRING ( 1 .. 18 );
```

In the SCHEMA_DEFINITION predefined package:

```
type IDENTIFIER is private;
```

For an <authorization identifier> a:

```
type AUTHORIZATION_IDENTIFIER_a is private;

function a return AUTHORIZATION_IDENTIFIER_a;

function a return AUTHORIZATION_IDENTIFIER_LIST;

type TABLE_NAME is private;
```

For a table t: – t functions are declared if and only if all <schema
– package>s referenced by the <Ada/SQL compilation unit>
– declare exactly one table named t

```
function t return TABLE_NAME;

function t return FROM_CLAUSE;

type TABLE_NAME_t is private;

function t return TABLE_NAME_t;

type COLUMN_NAME_t is private;
```

For a table t with <authorization identifier> a (all other tables with <authorization identifier> a are similarly included in the record type declarations):

```
type TABLE_NAME_UNTYPED_a is
  record
    . . .
    t : TABLE_NAME;
    . . .
```

```
    end record;

  function a return TABLE_NAME_UNTYPED_a;

  type TABLE_NAME_TYPED_a is
    record
      . . .
      t : TABLE_NAME_t;
      . . .
    end record;

  function a return TABLE_NAME_TYPED_a;

  type FROM_CLAUSE_a is
    record
      . . .
      t : FROM_CLAUSE;
      . . .
    end record;

  function a return FROM_CLAUSE_a;
```

For a data type ct, used as the data type of a column within table t:

```
  type COLUMN_NAME_t_ct is private;
```

For a column of data type ct, with <column name> c, declared in table t:

```
  function c return COLUMN_NAME_t;

  function c return COLUMN_NAME_t_ct;

type CURSOR_NAME is private;

NULL_CURSOR_NAME : constant CURSOR_NAME;
```

Example

examples of names are used in various syntactic constructs containing them

Syntax Rules

1) A <table name> identifies a named table. References to <table name>s in this section also pertain to a <table name> represented in a <table name with optional column list> or an <underscored table name>.

2) If a <table name> does not contain an <authorization identifier>, then:

   a) If the <table name> is contained in a <schema>, then the <authorization identifier> specified as the <schema authorization identifier> of the <schema> is implicit.

b) If the <table name> is not contained in a <schema>, then exactly one distinct <library package name> contained in the <Ada/SQL compilation unit> containing the <table name> shall be the name of a <schema package> declaring a table with the <table identifier> used in the <table name>, and the <authorization identifier> of that <schema package> is implicit.

3) Two <table name>s are equal if and only if they have the same <table identifier> and the same <authorization identifier>, regardless of whether the <authorization identifier>s are implicit or explicit.

4) A <table name> is declared in a <table definition>.

5) An <Ada/SQL compilation unit> containing a <table name> shall also contain a <library package name> that is the name of a <schema package> containing a <table definition> declaring the <table name>, unless the <Ada/SQL compilation unit> is part of such a <schema package> itself.

6) An <authorization identifier> represents an authorization identifier.

7) A <database identifier> is declared as a <correlation name> for a particular table. The <correlation name> is associated with a particular instance of that table for a particular scope. The scope of a <correlation name> is either a <select statement>, <subquery>, or <query specification> (see 5.20, "<from clause>"). Scopes may be nested. *In different scopes, the same <correlation name>* may be associated with different instances of the same table.

8) A <column name> identifies a named column. An <identifier> is defined as a <column name> by a <table definition>. An <Ada/SQL compilation unit> containing a <table name> shall also contain a <library package name> that is the name of a <schema package> containing a <table definition> declaring the <column name>, unless the <Ada/SQL compilation unit> is part of such a <schema package> itself.

9) A <cursor name> identifies a cursor. A <cursor name> shall be the same as a <variable name> declared of type CURSOR_NAME.

10) A <package name> denotes one of the packages described in this specification, in accordance with Ada rules. Section 6.1.4 contains restrictions on the forms of <package name>s allowed in various contexts. A <non Ada/SQL package name> is the name of a package containing program text that is not relevant to Ada/SQL.

11) A <library package name> denotes an Ada library package, in accordance with Ada rules. A <non Ada/SQL library unit name> is the name of a library unit containing program text that is not relevant to Ada/SQL. A <package identifier> is declared as the name of a package.

12) A <type mark> denotes a data type or subtype, in accordance with Ada rules. A <type identifier> is declared as the name of a data type or subtype.

13) A <program object name> is the name of a variable (<variable name>) or named number (<named number name>), in accordance with Ada rules. A <simple variable name> is declared as the name of a variable.

General Rules

None.

Notes

1) Type DATABASE.USER_AUTHORIZATION_IDENTIFIER is used as the data type of the <keyword> USER.

2) Type IDENTIFIER refers to an <authorization identifier>, but is not named AUTHORIZATION_IDENTIFIER due to the syntax of <schema authorization clause>s. Its only use is in <authorization package>s and <schema authorization clause>s. No functions returning type IDENTIFIER are shown here as being effectively declared; instantiating AUTHORIZATION_IDENTIFIER within an <authorization package> performs the effective declaration.

3) The <table name> a.t effectively calls one of the a functions to return a value of a record type; then selects the t component of this value, which is the appropriately typed value representing the <table name>.

4) Ada/SQL names conform to ANSI SQL names. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	5
SR2	SR2	6
SR3	SR3	-
SR4	SR4	7
SR5	SR5	8
SR6	SR6	-
SR7	SR7	9
SR8	SR8	10
SR9	—	11
SR10	SR9	12
SR11-SR12	—	11
—	SR10-SR13	13

5) ANSI SQL <table name> syntax has been mirrored in Ada/SQL for most contexts. There are several, noted in Ada/SQL SR1, however, where Ada syntax forces modifications to ANSI SQL syntax for Ada/SQL.

6) ANSI SQL and Ada/SQL identically interpret <table name>s contained in a <schema>. For <table name>s not contained in a <schema>, however, ANSI SQL uses the <authorization identifier> of the containing <module>, if the <table name> does not contain an <authorization identifier>. Since Ada/SQL does not have a <module> concept, this approach cannot be used. Instead, Ada/SQL requires that a <table name> without an <authorization identifier> unambiguously denote a table.

7) Tables are declared by both <table definition>s and <view definition>s in ANSI SQL. Ada/SQL requires a <table definition> for both base tables and viewed tables, so every <table name> is declared in a <table definition>.

8) Ada/SQL SR5 relates <table name> references to the <schema package> that declares the <table name>. Note that a <schema> may be divided up into several <schema package>s, in accordance with Ada separate compilation philosophy, and that an <Ada/SQL compilation unit> may reference only selected <schema package>s from a <schema>, using accepted modularity concepts.

9) ANSI SQL <correlation name>s are defined by their appearance in a <table reference>. In order to get similar Ada/SQL syntax, it is necessary to predefine <correlation name>s with <correlation name declaration>s. Each <correlation name> is associated with a particular table by a <correlation name declaration>.

10) In Ada/SQL, a <table definition> is required for both base tables and viewed tables. Hence, a <column name> is defined by its containing <table definition>s. In ANSI SQL, a <column name> can be defined in either a <table definition> or a <view definition>.

11) <module name>s, <procedure name>s, and <parameter name>s are not relevant to Ada/SQL.

12) In ANSI SQL, <cursor name>s are totally contained within a <module>; there is no need to link them to program variables. In Ada/SQL, program variables are used to represent cursors, and these program variables are strongly typed as CURSOR_NAMEs.

13) Various Ada constructs that are not relevant to ANSI SQL are described in Ada/SQL SR10-SR13.

5.5 <data type>

Function

Specify a subtype to be used in declaring a data type.

Format

<data type> ::=
  <character string type>
| <integer type>
| <floating point type>
| <enumeration type>

Effective Ada Declarations

None.

Example

```
type LAST_NAME is array ( 1 .. 10 ) of CHARACTER;
  -- <character string type>

type EMPLOYEE_COUNT is range  1 .. 10_000;
  -- <integer type>

type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;
  -- <floating point type>

type SUIT is ( CLUBS , DIAMONDS , HEARTS , SPADES );
  -- <enumeration type>
```

Syntax Rules

None.

General Rules

None.

Notes

1) The correspondence between Ada/SQL and ANSI SQL <data type>s is as follows:

ANSI SQL	Ada/SQL	See Notes
<character string type>	<character string type>	2
<exact numeric type>	<integer type>	3
<approximate numeric type>	<floating point type>	4
—	<enumeration type>	5

2) Ada/SQL <character string type>s are analogous to ANSI SQL <character string type>s. The components of Ada/SQL <character string type>s can be constrained to allow only certain <character>s. Although an Ada/SQL <character string type> may be unconstrained, allowing different strings of the same data type to be of different lengths, each column declared to be of that data type must be of a fixed, specified length.

3) Ada/SQL <integer type>s are analogous to a subset of ANSI SQL <exact numeric type>s, namely those with a <scale> of 0. Also, a <range constraint> may be placed on an Ada/SQL <integer type>. ANSI SQL <exact numeric type>s have a fixed decimal point location, indicated by their <scale>. It is therefore tempting to map them into Ada fixed point types. Such a mapping is not necessarily correct, however, because ANSI SQL <exact numeric type>s represent exact decimal values, while Ada fixed point types will represent approximations to decimal values unless the environment supports Ada length clauses for their SMALL attribute. Since not all environments support this, it was felt best to avoid defining a mapping that might appear intuitive to users and yet allow computation errors to occur. A later version of this standard may provide a way to map all ANSI SQL <exact numeric type>s to Ada and Ada/SQL, most likely as one or more generic packages providing an internal representation and all required operations on <exact numeric type>s. In the meantime, most computations for which <exact numeric type>s appear suited can instead be performed using <floating point type>s of sufficient accuracy.

4) Ada/SQL <floating point type>s are analogous to ANSI SQL <approximate numeric type>s, except that ANSI SQL expresses precision in terms of bits (binary digits), while Ada/SQL expresses accuracy in terms to decimal digits. Also, a <range constraint> may be placed on an Ada/SQL <floating point type>.

5) ANSI SQL has no analog of Ada/SQL's <enumeration type>s. Enumeration values may be represented in an SQL database as, for example, integers, but the ability to name such values with Ada/SQL <enumeration type>s is of value to software engineering.

5.5.1 <character string type>

Function

Specify a character string subtype.

Format

<character string type> ::=
    <unconstrained character string definition>
  | <constrained character string definition>

<unconstrained character string definition> ::=
    array (<index subtype definition>) **of** <component subtype indication>

<constrained character string definition> ::=
    array <index constraint> **of** <component subtype indication>

<index subtype definition> ::=
    <type mark> **range** <>

<component subtype indication> ::= <subtype indication>

<index constraint> ::= (<discrete range>)

<discrete range> ::=
    <discrete subtype indication>
  | <range>

<discrete subtype indication> ::= <subtype indication>

Effective Ada Declarations

In the DATABASE predefined package:

```
   MAX_CHARACTERS : constant := implementation_defined;
-- maximum number of characters in a <character string type> supported
-- by the Ada/SQL environment
```

Example

```
type UNCONSTRAINED_NAME is array ( POSITIVE range <> ) of CHARACTER;

type LAST_NAME is array ( 1 .. 10 ) of CHARACTER;

type HEX_CHARACTER is
     ( '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7',
       '8' , '9' , 'A' , 'B' , 'C' , 'D' , 'E' , 'F' );

type HASH_CODE_INDEX is range 1 .. 8;

type HASH_CODE is array ( HASH_CODE_INDEX ) of HEX_CHARACTER;
```

Syntax Rules

1) The <type mark> of an <index subtype definition> shall denote an integer subtype containing at least one value within its range.

2) A <component subtype indication> shall denote a character subtype containing at least one value, with values ordered according to the ASCII collating sequence.

3) The subtype defined by an <index constraint> shall be an integer subtype containing at least one value but not more than DATABASE.MAX_CHARACTERS values within its range.

General Rules

1) A <component subtype indication> or a <discrete subtype indication> defines the same subtype as does its contained <subtype indication>.

2) Case:

   a) A <discrete range> that contains a <discrete subtype indication> defines the same subtype as does the <discrete subtype indication>.

   b) A <discrete range> not containing a <discrete subtype indication> defines a subtype of the data type of the contained <range>, with lower and upper bounds given by the two contained <value specification>s, respectively.

3) An <index constraint> defines the same subtype as does its contained <discrete range>.

4) The *component subtype* of a character string subtype or object declared as a <character string type> is that defined by the contained <component subtype indication>.

5) Case:

   a) The *index subtype* of a character string data type declared with an <unconstrained character string definition> is that denoted by the <type mark> of the contained <index subtype definition>. The compound delimiter <> (called a *box*) of an <index subtype definition> stands for an undefined range (different objects of the unconstrained character string data type need not have the same bounds).

   b) The *index subtype* of a character string data type or object declared with a <constrained character string definition> is that defined by the contained <discrete range>.

Notes

1) The Format and Rules for the Ada/SQL <character string type> are patterned after the Ada array_type_definition, suitably restricted apropos of ANSI SQL capabilities.

2) The original Ada/SQL definition allowed null index ranges, stating that columns declared of character string types with null index ranges would only be permitted to store null values. But such a concept seems so useless that it is simply not allowed in this standard.

3) Release 1 implementations impose a tighter constraint on <component subtype indication> than does SR2, requiring that it denote a subtype of the predefined STANDARD.CHARACTER type or a type derived therefrom. Release 1 implementations also do not permit a <component subtype indication> to contain a <constraint>.

4) The reason SR2 requires that the character values of a component subtype be ordered according to the ASCII collating sequence is so that character strings will have the same ordering under both Ada and ANSI SQL semantics.

5) Release 1 implementations do not provide the <named number> DATABASE.MAX_CHARACTERS.

6) <range> is the only form of <discrete range> allowed by Release 1 implementations within a <subtype indication>.

5.5.2 <integer type>

Function

Specify an integer subtype.

Format

<integer type> ::=
  <range constraint>

Effective Ada Declarations

```
In the DATABASE predefined package:

  MIN_INT : constant := implementation_defined;
    -- smallest (most negative) integer value supported by the Ada/SQL
    --  environment

  MAX_INT : constant := implementation_defined;
    -- largest (most positive) integer value supported by the Ada/SQL
    --  environment
```

Example

```
type HASH_CODE_INDEX is range 1 .. 8;
```

Syntax Rules

1) The <range constraint> shall have integer bounds, with the lower bound not less than DATABASE.MIN_INT and the upper bound not greater than DATABASE. MAX_INT.

General Rules

1) The range of values defined by an <integer type> is the same as that defined by the <range constraint>.

Notes

1) The Format and Rules for the Ada/SQL <integer type> are patterned after the Ada integer_type_definition, suitably restricted apropos of ANSI SQL capabilities.

2) Release 1 implementations do not provide the <named number>s DATABASE. MIN_INT and DATABASE.MAX_INT.

5.5.3 <floating point type>

Function

Specify a floating point subtype.

Format

<floating point type> ::=
  <floating point constraint>

<floating point constraint> ::=
  <floating accuracy definition> [<range constraint>]

<floating accuracy definition> ::=
  digits <value specification>

Effective Ada Declarations

In the DATABASE predefined package:

```
MAX_DIGITS : constant := implementation_defined;
  -- maximum number of digits that can be specified in a
  --  <floating accuracy definition>

DOUBLE_PRECISION_SAFE_LARGE : constant := implementation-defined;
  -- largest positive floating point number permitted
```

Example

```
type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

type MOST_PRECISE_TYPE is digits DATABASE.MAX_DIGITS;
```

Syntax Rules

1) The <value specification> shall be of an integer data type, shall be positive (nonzero), and shall not be greater than DATABASE.MAX_DIGITS.

2) The <value specification> shall not contain a <program object name> other than a <named number>, an <indicator specification>, an <Ada type conversion>, or a CONVERT_TO operation.

3) The <range constraint> (if any) shall have floating point bounds.

4) The bounds of the <range constraint> (if any) shall not exceed DATABASE.DOUBLE_PRECISION_SAFE_LARGE in absolute value.

5) Let B be the integer next above

(DATABASE.MAX_DIGITS * log(10)/log(2)) + 1.

(B is the smallest number of significant bits required in a binary representation of a number with DATABASE.MAX_DIGITS decimal digits, such that the relative precision of the binary form is no less than that of the decimal form.) DATABASE.DOUBLE_PRECISION_SAFE_LARGE shall be no less than

2**(4*B) - 2**(3*B).

General Rules

1) Numbers represented in the <floating point type> shall retain at least the number of significant decimal digits given by the <value specification>.

2) The range of values defined by a <floating point type> is the same as that defined by the <floating point constraint>.

3) Case:

   a) If a <floating point constraint> contains a <range constraint>, then the range of values defined by that <floating point constraint> is the same as that defined by the <range constraint>.

   b) If a <floating point constraint> does not contain a <range constraint>, then:

      Case:

      i) If the <floating point constraint> is contained within a <full type declaration>, then the range of values it defines is implementation-dependent, and may be different for Ada operations and for database operations.

         1) An Ada implementation selects a representation to be used for numbers satisfying the <floating point constraint>, and the range of values is determined by the representation selected.

         2) Likewise, an Ada/SQL implementation selects a representation to be used for numbers satisfying the <floating point constraint> and the range of values is determined by the representation selected. Since the representation selected by the Ada implementation may not be the same as that selected by the Ada/SQL implementation (database management system), the ranges of values defined may also differ. *In all cases (including the representations selected for integer data types), the base type of a data type is considered to contain the range of values determined by the Ada/SQL implementation. Where it is necessary to distinguish the range of values selected by an Ada implementation, that range is explicitly attributed to the Ada base type.*

3) Let D be the value of the <value specification>. Let B be the integer next above (D * log(10)/log(2)) + 1. (B is the smallest number of significant bits required in a binary representation of a number with D decimal digits, such that the relative precision of the binary form is no less than that of the decimal form.) The range of values of the base type shall include at least -R..R, where

R = 2**(4*B) - 2**(3*B).

ii) If the <floating point constraint> is contained within a <subtype indication>, then the range of values defined by that <floating point constraint> is the same as that denoted by the <type mark> contained in the <subtype indication>.

4) The accuracy defined by a <floating point constraint> is the value of the contained <value specification>.

Notes

1) The Format and Rules for the Ada/SQL <floating point type> are patterned after the Ada floating_point_constraint, suitably restricted apropos of ANSI SQL capabilities.

2) Release 1 implementations do not provide the <named number>s DATABASE. MAX_DIGITS or DATABASE.DOUBLE_PRECISION_SAFE_LARGE.

3) SR2 ensures that the <value specification> is a meaningful static Ada simple_expression, as required by Ada syntax and semantics.

4) SR4 and SR5 ensure that the range of floating point numbers supported by the database includes the Ada model numbers for the maximum number of significant digits supported by the database.

5) SDL syntax permits the declaration of a floating point data type only in terms of its accuracy (minimum number of significant digits), without specifying an allowable range of values. GR3b.i requires that the range of such a data type must include at least the Ada model numbers for the declared accuracy.

5.5.4 <enumeration type>

Function

Specify an enumeration subtype.

Format

<enumeration type> ::=
(<enumeration literal specification>
[{ , <enumeration literal specification> } ...])

<enumeration literal specification> ::=
<simple enumeration literal>

Effective Ada Declarations

None.

Example

```
type HEX_CHARACTER is
     ( '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7',
       '8' , '9' , 'A' , 'B' , 'C' , 'D' , 'E' , 'F' );
```

Syntax Rules

1) The <enumeration literal specification>s contained in an <enumeration type> shall be distinct.

2) An <enumeration type> is said to be a *character type* (or *character subtype*, *character data type*, as appropriate)if all of its <simple enumeration literal>s are <character literal>s.

General Rules

1) Each <enumeration literal specification> declares its contained <simple enumeration literal>. Note that the same <simple enumeration literal> can be declared for several different <enumeration type>s. If an <enumeration literal> occurs in a context that does not otherwise suffice to determine the data type of the <enumeration literal>, then an <Ada type qualification> is one way to resolve the ambiguity.

2) Each <simple enumeration literal> yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding *position numbers*. The position number of the value of the first listed <simple enumeration literal> is zero; the position number for each other <simple enumeration literal> is one more than for its predecessor in the list.

Notes

1) The Format and Rules for the Ada/SQL <enumeration type> are patterned after the Ada enumeration_type_definition, suitably restricted apropos of ANSI SQL capabilities.

2) The definition of character type given in SR2 differs from that of Ada, which considers an enumeration type to be a character type if at least one of its enumeration literals is a character literal. The Ada/SQL definition is used to indicate the allowable components of character strings.

5.5.5 <subtype indication>

Function

Specify a subtype of a data type.

Format

<subtype indication> ::=
  <type mark> [<constraint>]

<constraint> ::=
  <range constraint> | <index constraint> | <floating point constraint>

Effective Ada Declarations

None.

Example

```
type HEX_CHARACTER is
     ( '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7',
       '8' , '9' , 'A' , 'B' , 'C' , 'D' , 'E' , 'F' );

subtype OCTAL_CHARACTER is HEX_CHARACTER range '0' .. '7';

type UNCONSTRAINED_NAME is array ( POSITIVE range <> ) of CHARACTER;

subtype CONSTRAINED_NAME is UNCONSTRAINED_NAME ( 1 .. 20 );

type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

subtype BOSS_RELATIVE_SALARY is EMPLOYEE_SALARY
 range 80_000.00 .. 99_999.99;
```

Syntax Rules

1) If a <constraint> is contained within a <subtype indication>, then:

   Case:

   a) If the <type mark> denotes a character string subtype, then:

      i) The <type mark> shall denote an unconstrained character string subtype.

      ii) The <constraint> shall contain an <index constraint>.

      iii) The subtype defined by the <index constraint> shall be of the same data type as the index subtype of the character string subtype denoted by the <type mark>.

iv) The bounds of the subtype defined by the <index constraint> shall belong to the index subtype of the character string subtype denoted by the <type mark>.

b) If the <type mark> denotes an integer or an enumeration subtype, then the <constraint> shall immediately contain a <range constraint>.

c) If the <type mark> denotes a floating point subtype, then:

i) The <constraint> shall immediately contain either a <range constraint> or a <floating point constraint>.

ii) If the <constraint> contains a <floating point constraint>, then the value of the <value specification> contained in the <floating accuracy definition> of the <floating point constraint> shall not exceed the accuracy of the subtype denoted by the <type mark>.

2) If a <range constraint> is contained within a <subtype indication>, then:

a) The data type of the <range constraint> shall be the same as that denoted by the <type mark>.

b) The bounds of the <range constraint> shall belong to the subtype denoted by the <type mark>.

General Rules

1) A <subtype indication> defines a subtype of the data type denoted by the <type mark>.

2) Case:

a) If no <constraint> appears within the <subtype indication>, then the subtype defined is the same as that denoted by the <type mark>.

b) If a <constraint> appears within the <subtype indication>, then:

Case:

i) If the <type mark> denotes a character string data type, then:

1) The subtype defined is a constrained character string subtype.

2) The component subtype of the subtype defined is the same as that of the subtype denoted by the <type mark>.

3) The index subtype of the subtype defined is the subtype defined by the <index constraint>.

ii) If the <type mark> denotes an integer data type, then the range of values of the subtype defined is that defined by the <range constraint>.

iii) If the <type mark> denotes a floating point data type, then:

Case:

1) If the <subtype indication> contains a <floating point constraint>, then the accuracy and range of values of the subtype defined are those defined by the <floating point constraint>.

2) If the <subtype indication> does not contain a <floating point constraint> (contains only a <range constraint>), then:

a) The accuracy of the subtype defined is the same as that of the subtype denoted by the <type mark>.

b) The range of values of the subtype defined is that defined by the <range constraint>.

iv) If the <type mark> denotes an enumeration data type, then the <enumeration literal>s of the subtype defined are those of the subtype denoted by the <type mark> that lie within the range of values defined by the <range constraint>.

Notes

1) The Format and Rules for the Ada/SQL <subtype indication> are patterned after the Ada subtype_indication, suitably restricted apropos of ANSI SQL capabilities.

2) Compliance with SRs 1a.iv and 2b is checked by Ada implementations at runtime, and so the rules might be expressed as GRs for Ada/SQL. The conditions checked may be dynamic in Ada; the semantics of interfacing with ANSI SQL require that these conditions be known at compile time in Ada/SQL. An Ada/SQL implementation that reads source code can therefore verify compliance with these rules. Certain violations of these rules will cause Ada to raise CONSTRAINT_ERROR at runtime, but specification of when CONSTRAINT_ERROR is raised would require describing the elaboration of Ada declarations, which is beyond the scope of Ada/SQL.

3) The specification of SRs 1a.iv and 2b is simplified by not allowing null ranges in Ada/SQL. The illegal Ada/SQL use of null ranges will (in the absence of other errors) not be caught at runtime, since null ranges are legal in Ada. Again, an Ada/SQL implementation that reads source code could catch such errors. It should be noted that an Ada/SQL implementation would almost certainly have to read the SDL source code, which is where most <subtype indication>s would typically appear. <subtype indication>s can also appear in <variable declaration>s, however, which would

conceivably not have to be read by an Ada/SQL implementation. Using a variable declared in violation of these rules, but not of Ada rules, would most likely cause CONSTRAINT_ERROR to be raised, not at its declaration, but at its use in an Ada/SQL statement.

5.5.6 <range constraint>

Function

Specify a range of values.

Format

<range constraint> ::=
  range <range>

<range> ::=
  <value specification> .. <value specification>

Effective Ada Declarations

None.

Example

```
type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;
```

Syntax Rules

1) Each <value specification> shall be of an integer, floating point, or enumeration data type.

   Case:

   a) If the first <value specification> is of an integer data type, then the second <value specification> shall also be of an integer data type.

   b) If the first <value specification> is of a floating point data type, then the second <value specification> shall also be of a floating point data type.

   c) If the first <value specification> is of an enumeration data type, then the second <value specification> shall also be of an enumeration data type.

2) The <value specification>s shall not contain a <program object name> other than a <named number>, an <indicator specification>, an <Ada type conversion>, or a CONVERT_TO operation.

3) The value of the first <value specification> shall not be greater than the value of the second <value specification>.

4) Case:

   a) If a <range> is immediately contained in a <subtype indication>, then it is of a defined data type.

b) If a <range> is contained in an <index constraint> of a <character string type>, but is not contained in a <subtype indication>, then it is of a defined data type.

c) If a <range> is contained in an <integer type> or a <floating point type>, then it is not of a defined data type.

5) If a <range> is of a defined data type, then the data types of both <value specification>s contained within it shall be the same.

Case:

a) If both <value specification>s are of known data types, then both data types shall be the same and the data type of the <range> is the same as that data type.

b) If one <value specification> is of a known data type, then the value of the other <value specification> shall be a value of that data type, and the data type of the <range> is the same as that data type.

c) If neither <value specification> is of a known data type, then:

Case:

i) If the <range> is immediately contained in a <subtype indication>, then:

Case:

1) If the <type mark> of the <subtype indication> denotes a character string subtype, then the value of each <value specification> shall be a value of the index subtype of that character string subtype, and the data type of the <range> is the data type of the index subtype.

2) If the <type mark> of the <subtype indication> denotes an integer, floating point, or enumeration data type, then the value of each <value specification> shall be a value of that data type, and the data type of the <range> is the data type denoted by the <type mark>.

ii) If the <range> is contained in an <index constraint> of a <character string type>, but is not contained in a <subtype indication>, then:

1) Each <value specification> shall consist of only a single <integer literal> or a single integer <named number>.

2) The data type of the <range> is STANDARD.INTEGER.

6) If a <range> is of a defined data type, then a <range constraint> containing it (if any) is of the same data type.

General Rules

1) A <range> defines a range of values. The first <value specification> specifies the lower bound of the range; the second <value specification> specifies the upper bound of the range. A value between the lower and upper bounds, inclusive, belongs to the range.

2) A <range constraint> defines the same range of values as its contained <range>, with the same bounds.

3) A <range> that is contained within an <index constraint> of a <character string type>, but that is not contained in a <subtype indication>, defines an integer subtype. The data type and bounds of the subtype are those of the <range>. (The subtype is used as the index subtype of the <character string type>.)

Notes

1) The Format and Rules for the Ada/SQL <range constraint> are patterned after the Ada range_constraint, suitably restricted apropos of ANSI SQL capabilities.

2) SR2 ensures that the <value specification>s are meaningful static Ada simple_expressions. Although not all contexts for <range> strictly require static expressions, requiring all <range>s to be static simplifies their specification and does not cost any capability with respect to ANSI SQL.

3) SR3 explicitly prohibits null <range>s. Although permitted in earlier Ada/SQL specifications, the utility of null <range>s appears so negligible that the simplification gained by prohibiting them far outweighs any loss of capability.

5.6 <value specification> and <target specification>

Function

Specify one or more values or variables.

Format

```
<value specification> ::=
    [ + | - ] <value specification term>
  | <value specification> + <value specification term>
  | <value specification> - <value specification term>

<value specification term> ::=
    <value specification factor>
  | <value specification term> * <value specification factor>
  | <value specification term> / <value specification factor>

<value specification factor> ::= <value specification primary>

<value specification primary> ::=
    <variable specification>
  | <literal>
  | USER
  | <Ada type qualification>
  | <Ada type conversion>
  | [ CONVERT_TO . <library package name> . <type identifier> ]
    ( <value specification> )

<variable specification> :=
    <program object name>
  | <indicator specification>

<indicator specification> ::=
    INDICATOR ( <value specification> [ , <indicator value> ] )

<indicator value> ::=
    <value specification>
  | NOT_NULL
  | NULL_VALUE

<Ada type qualification> ::=
    <type mark> ' ( <value specification> )

<Ada type conversion> ::=
    <type mark> ( <value specification> )

<target specification> ::=
    <program variable> [ , <last variable> ] [ , <indicator variable> ]
```

<program variable> ::= <out variable>

<last variable> ::= <out variable>

<indicator variable> ::= <out variable>

<out variable> ::=
  <variable name>
| <type mark> (<variable name>)

Effective Ada Declarations

```
type VALUE_SPECIFICATION is private;

type VALUE_SPECIFICATION_INTEGER is private;

type VALUE_SPECIFICATION_FLOATING is private;

type VALUE_SPECIFICATION_STRING is private;

type INDICATOR_VARIABLE is ( NULL_VALUE , NOT_NULL );

For a program data type ct:

  type VALUE_SPECIFICATION_ct is private;
```

For an enumeration data type ct that is not derived from another enumeration type (an ultimate parent type):

```
  type VALUE_SPECIFICATION_ENUMERATION_ct is private;
```

For a program data type ct:

```
  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_SPECIFICATION_ct;

  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_SPECIFICATION;

  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_EXPRESSION_ct;

  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_EXPRESSION;
```

For an integer program data type ct:

```
  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_SPECIFICATION_INTEGER;

  function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
   return VALUE_EXPRESSION_INTEGER;
```

For a floating point program data type ct:

```
function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION_FLOATING;

function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_FLOATING;
```

For a character string program data type ct:

```
function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION_STRING;

function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_STRING;
```

For an enumeration program data type ct with ultimate parent type pt:

```
function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION_ENUMERATION_pt;

function INDICATOR ( VALUE : ct ; IND : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_ENUMERATION_pt;

-- VALUE_SPECIFICATION_DATABASE_USER_AUTHORIZATION_IDENTIFIER is defined for
-- predefined type DATABASE.USER_AUTHORIZATION_IDENTIFIER in accordance with
-- the above - For a program data type ct:
--   type VALUE_SPECIFICATION_ct is private;

-- VALUE_EXPRESSION_DATABASE_USER_AUTHORIZATION_IDENTIFIER is defined for
-- predefined type DATABASE.USER_AUTHORIZATION_IDENTIFIER in accordance with
-- 5.9 - For a program data type ct:
--   type VALUE_EXPRESSION_ct is private;

function USER
 return VALUE_SPECIFICATION_DATABASE_USER_AUTHORIZATION_IDENTIFIER;

function USER return VALUE_SPECIFICATION_STRING;

function USER return VALUE_SPECIFICATION;

function USER
 return VALUE_EXPRESSION_DATABASE_USER_AUTHORIZATION_IDENTIFIER;

function USER return VALUE_EXPRESSION_STRING;

function USER return VALUE_EXPRESSION;

type USER_VALUE_SPECIFICATION is private;

function USER return USER_VALUE_SPECIFICATION;
```

```
function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION_DATABASE_USER_AUTHORIZATION_IDENTIFIER;

function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION;

function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_DATABASE_USER_AUTHORIZATION_IDENTIFIER;

function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL ) return VALUE_EXPRESSION;

function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION_STRING;

function INDICATOR
         ( VALUE : USER_VALUE_SPECIFICATION;
           IND   : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_STRING;
```

For an integer program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
    . . .
    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION_dt;

    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION;

    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION_INTEGER;
    . . .
  end p;
   . . .
end CONVERT_TO;
```

For a floating point program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : VALUE_SPECIFICATION_FLOATING )
     return VALUE_SPECIFICATION_dt;

    function ct ( LEFT : VALUE_SPECIFICATION_FLOATING )
     return VALUE_SPECIFICATION;

    function ct ( LEFT : VALUE_SPECIFICATION_FLOATING )
     return VALUE_SPECIFICATION_FLOATING;

    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION_dt;

    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION;

    function ct ( LEFT : VALUE_SPECIFICATION_INTEGER )
     return VALUE_SPECIFICATION_FLOATING;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

For a character string program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : VALUE_SPECIFICATION_STRING )
     return VALUE_SPECIFICATION_dt;

    function ct ( LEFT : VALUE_SPECIFICATION_STRING )
     return VALUE_SPECIFICATION;

    function ct ( LEFT : VALUE_SPECIFICATION_STRING )
     return VALUE_SPECIFICATION_STRING;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

For an enumeration program subtype ct defined in library package p, of data type dt with ultimate parent type pt (ct may be the same as dt, and dt may be the same as pt):

```
package CONVERT_TO is
   . . .
```

```
    package p is
      . . .
      function ct ( LEFT : VALUE_SPECIFICATION_ENUMERATION_pt )
       return VALUE_SPECIFICATION_dt;

      function ct ( LEFT : VALUE_SPECIFICATION_ENUMERATION_pt )
       return VALUE_SPECIFICATION;

      function ct ( LEFT : VALUE_SPECIFICATION_ENUMERATION_pt )
       return VALUE_SPECIFICATION_ENUMERATION_pt;
       . . .
    end p;
     . . .
  end CONVERT_TO;
```

Example

```
NEW_EMPLOYEE_NAME    : EMPLOYEE_NAME;
NEW_EMPLOYEE_SALARY  : HOURLY_WAGE;
SALARY_IS_KNOWN      : INDICATOR_VARIABLE;
CURRENT_EMPLOYEE,
HIS_MANAGER          : EMPLOYEE_NAME;
HIS_SALARY           : HOURLY_WAGE;
CURSOR               : CURSOR_NAME;
EMPLOYEE_LAST,
MANAGER_LAST         : NATURAL;
SALARY_INDICATOR,
MANAGER_INDICATOR    : INDICATOR_VARIABLE;
 . . .
INSERT_INTO ( EMPLOYEE ( NAME & SALARY & MANAGER ),
 VALUES <= NEW_EMPLOYEE_NAME
       and CONVERT_TO.EXAMPLE_TYPES.EMPLOYEE_SALARY
           ( INDICATOR ( 2080.0 * NEW_EMPLOYEE_SALARY , SALARY_IS_KNOWN ) )
       and USER );

-- variations: INDICATOR ( - 2080.0 / NEW_EMPLOYEE_SALARY , NOT_NULL )
               INDICATOR ( + 2080.0 + NEW_EMPLOYEE_SALARY , NULL_VALUE )
               INDICATOR ( ( 2080.0 - NEW_EMPLOYEE_SALARY ) );
 . . .
INSERT_INTO ( EMPLOYEE ( NAME & SALARY & MANAGER ),
 VALUES <= NEW_EMPLOYEE_NAME
       and EMPLOYEE_SALARY ( 2080.0 * NEW_EMPLOYEE_SALARY ),
       and USER );

-- variation: EMPLOYEE_SALARY ( HOURLY_WAGE'(2080.0) * NEW_EMPLOYEE_SALARY )
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( NAME & SALARY & MANAGER,
  FROM  => EMPLOYEE ) );
 . . .
FETCH ( CURSOR );
INTO ( CURRENT_EMPLOYEE , EMPLOYEE_LAST );
```

```
INTO ( EMPLOYEE_SALARY ( HIS_SALARY ) , SALARY_INDICATOR );
  -- variation: INTO ( EMPLOYEE_SALARY ( HIS_SALARY ) );
INTO ( HIS_MANAGER , MANAGER_LAST , MANAGER_INDICATOR );
```

Syntax Rules

1) A <value specification> specifies a value that is not selected from a table.

2) A <variable specification> identifies a named host object or a named host object and an indicator value. The data type of the indicator value shall be INDICATOR_VARIABLE.

3) A <target specification> specifies a variable that can be assigned a value.

4) The data type of USER is DATABASE.USER_AUTHORIZATION_IDENTIFIER.

5) A <value specification term>, <value specification>, or <value specification factor> that is an operand to one of the arithmetic operators shall not contain an <indicator specification>.

6) If a <value specification primary> not contained in an <indicator value> is of a character string or an enumeration data type, then the <value specification> containing it shall not include any arithmetic operators. The data type of the result is the same as that of the <value specification primary>.

7) A <value specification> not contained in an <indicator specification>, an <Ada type qualification>, an <Ada type conversion>, an <in value list>, or an <insert value list> shall not contain any arithmetic operators.

8) The data type of the result of a monadic arithmetic operator is the same as the data type of the (integer or floating point) <value specification term> to which it is applied.

9) Case:

   a) If both operands of a dyadic arithmetic operator are of a universal data type, then:

      Case:

      i) If both operands are of the same universal data type (universal integer or universal floating point), then the data type of the result is the same as that of the operands.

      ii) If one operand is of the universal integer data type and the other operand is of the universal floating point data type, then the result is of the universal floating point data type, and one of the following shall be true:

         1) The operator shall be multiplication, or

2) The operator shall be division, and the right operand shall be the one of the universal integer data type.

b) If either operand of a dyadic arithmetic operator is not of a universal data type, then both operands shall be of the same data type. The data type of the result is the same as that of the operands.

10) The data type of an <indicator specification> is that of its <value specification>.

11) An <indicator specification> shall not contain an <indicator specification> or a CONVERT_TO operation.

12) The <value specification> of an <indicator specification> shall contain at least one of the following: the <key word> USER, a <program object name> other than a <named number>, an <Ada type qualification>, or an <enumeration literal> which is a literal of exactly one enumeration data type declared in a <schema package> or the predefined Ada/SQL environment.

13) A <value specification> used as an <indicator value> shall be of data type INDICATOR_VARIABLE.

14) The data type of the result of an <Ada type qualification> or an <Ada type conversion> is that denoted by the <type mark>, and shall be an integer, floating point, character string, or enumeration data type.

15) The <value specification> of an <Ada type qualification> or an <Ada type conversion> shall not contain an <indicator specification> or the <key word> USER.

16) The <value specification> of an <Ada type conversion> shall contain at least one of the following: a <program object name> other than a <named number>, an <Ada type qualification>, or an <enumeration literal> which is a literal of exactly one enumeration data type declared in a <schema package> or the predefined Ada/SQL environment.

17) The <value specification> of an <Ada type qualification> shall be of the data type denoted by the <type mark>.

18) Case:

a) If the <type mark> of an <Ada type conversion> denotes an integer or floating point data type, then the <value specification> of that <Ada type conversion> shall be of an integer or floating point data type.

b) If the <type mark> of an <Ada type conversion> denotes a character string data type, then the <value specification> of that <Ada type conversion> shall be of a character string data type such that the component data types of both character string data types are the same.

c) If the <type mark> of an <Ada type conversion> denotes an enumeration data type, then the <value specification> of that <Ada type conversion> shall be of an enumeration data type such that both enumeration data types have the same ultimate parent type.

19) The data type of the result of a CONVERT_TO is that denoted by the <type identifier>, and shall be an integer, floating point, character string, or enumeration data type.

Case:

a) If the <library package name> is STANDARD, then the <type identifier> shall be declared within the STANDARD Ada/SQL predefined environment.

b) If the library package denoted by the <library package name> is part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within that library package.

c) If the library package denoted by the <library package name> is not part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within the ADA_SQL nested package of that library package.

20) The <value specification> operand of a CONVERT_TO shall contain at least one of the following: an <indicator specification> or the <key word> USER.

21) A <value specification> not contained in an <in value list>, an <insert value list>, or a <like predicate> shall not contain a CONVERT_TO operation.

22) Case:

a) If the <type identifier> of a CONVERT_TO denotes an integer data type, then the <value specification> operand of the CONVERT_TO shall be of an integer data type.

b) If the <type identifier> of a CONVERT_TO denotes a floating point data type, then the <value specification> operand of the CONVERT_TO shall be of an integer or a floating point data type.

c) If the <type identifier> of a CONVERT_TO denotes a character string data type, then the <value specification> operand of the CONVERT_TO shall be of a character string data type.

d) If the <type identifier> of a CONVERT_TO denotes an enumeration data type, then the <value specification> operand of the CONVERT_TO shall be of an enumeration data type such that both enumeration data types have the same ultimate parent type.

23) The data type of a <target specification> is that of its <program variable>, and shall be an integer, floating point, character string, or enumeration data type.

24) A <target specification> shall contain a <last variable> if and only if its <program variable> is of a character string data type. The data type of the <last variable> shall be the same as that of the index type of the <program variable>.

25) The data type of an <indicator variable> shall be INDICATOR_VARIABLE.

26) If an <out variable> contains a <type mark>, then the <type mark> shall denote a data type.

Case:

a) If the data type denoted by the <type mark> is declared with simple name ct in the STANDARD Ada/SQL predefined environment, then the form of the <type mark> shall be either "STANDARD.ct" or "ct".

b) If the data type denoted by the <type mark> is declared with simple name ct in a library package p that is part of the Ada/SQL predefined environment, then the form of the <type mark> shall be either "p.ct" or "ct".

c) If the data type denoted by the <type mark> is declared in a <schema package> p, with a <type declaration> containing <type identifier> ct, then the form of the <type mark> shall be either "p.ADA_SQL.ct" or "ct".

27) If an <out variable> contains a <type mark>, then:

Case:

a) If the <type mark> denotes an integer or floating point data type, then the variable denoted by the <variable name> shall be of an integer or floating point data type.

b) If the <type mark> denotes a character string data type, then the variable denoted by the <variable name> shall also be of a character string data type such that the component data types of both character string data types are the same.

c) If the <type mark> denotes an enumeration data type, then the variable denoted by the <variable name> shall be of an enumeration data type such that both enumeration data types have the same ultimate parent type.

28) Case:

a) If an <out variable> contains a <type mark>, then:

Case:

i) If the <type mark> does not denote an unconstrained character string subtype, then the subtype of the <out variable> is that denoted by the <type mark>.

ii) If the <type mark> denotes an unconstrained character string subtype, then the subtype of the <out variable> is that denoted by the <type mark>, further constrained with the actual index bounds of the variable denoted by the <variable name>.

b) If an <out variable> does not contain a <type mark>, then the subtype of the <out variable> is that of its <variable name>.

General Rules

1) If arithmetic operators are not specified, then the result of the <value specification> is the value of the specified <value specification primary>.

2) The monadic arithmetic operators + and - specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand. Except where the result is of a universal data type, the result of a monadic operator shall belong to the base type of its operand; otherwise the program causing the <value specification> to be evaluated is erroneous.

3) The dyadic arithmetic operators +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. A divisor shall not be 0; otherwise, the DATA_EXCEPTION exception is raised. Except where the result is of a universal data type, the result of a dyadic arithmetic operator shall belong to the base type of its operands; otherwise, the program causing the <value specification> to be evaluated is erroneous.

4) All arithmetic operators shall yield mathematically correct results.

a) The result of integer operations other than division shall be exact.

b) The result of integer division shall be truncated toward 0 to the nearest integer.

c) The result of floating point operations shall be correct to the accuracy of the data type of the result.

5) Expressions within parentheses are evaluated first and when the order of evaluation is not specified by parentheses, multiplication and division are applied before monadic operators, monadic operators are applied before addition and subtraction, and operators at the same precedence level are applied from left to right.

6) The value of a <variable specification> that is a <program object name> is the value of the program object denoted by the <program object name>. The value of a program object shall be defined at the time of its evaluation in an <SQL statement>; otherwise, the execution of the program causing the <variable specification> to be evaluated is erroneous.

7) If an <indicator specification> contains an <indicator value> that is "NULL_VALUE" or a <value specification> evaluating to "NULL_VALUE", then the value specified by the <indicator specification> is null. Otherwise, the value specified by an <indicator specification> is the value of

its <value specification>.

8) The value specified by a <literal> is the value represented by that <literal>.

9) The value specified by USER is equal to the implicit <authorization identifier> that has been assigned to the execution of the program causing the USER <value specification> to be evaluated.

10) The result of an <Ada type qualification> is the value of its <value specification>, typed according to the <type mark>. This value shall belong to the subtype denoted by the <type mark>; otherwise, the CONSTRAINT_ERROR exception is raised.

11) The result of an <Ada type conversion> is the value of its <value specification>, typed according to the <type mark>. If the result of the <value specification> does not belong to the subtype denoted by the <type mark>, then the CONSTRAINT_ERROR exception is raised.

    a) Conversion of an integer value to an integer value shall be exact.

    b) Conversion of a floating point value to an integer value shall round to the nearest integer; a result halfway between two integers (to the accuracy of the floating point data type) may be rounded either up or down. Any program whose effect depends on the direction of rounding of values halfway between two integers is erroneous.

    c) Conversion of an integer or a floating point value to a floating point value shall retain at least the accuracy of the subtype denoted by the <type mark>.

    d) Conversion of a character string value to a character string value shall be as follows:

        i) The data type of the <value specification> and the data type denoted by the <type mark> shall both have component subtypes allowing the same range of characters; otherwise, the CONSTRAINT_ERROR exception is raised.

        ii) If the <type mark> denotes an unconstrained character string subtype then the index bounds of the result are the same as the index bounds of the <value specification>, converted to the index data type of the unconstrained character string subtype denoted by the <type mark>. If the <value specification> is not a null character string, then the index bounds shall belong to the index subtype; otherwise, the CONSTRAINT_ERROR exception is raised.

        iii) If the <type mark> denotes a constrained character string subtype, then the number of characters in the character string subtype shall be the same as the number of characters in the <value specification>; otherwise, the CONSTRAINT_ERROR exception is raised.

        iv) Successive characters in the result are set to successive characters in the <value specification>.

e) Conversion of an enumeration value to another enumeration value shall be according to matching enumeration literals.

12) The result of a CONVERT_TO with a <value specification> equal to the null value is the null value. If the subtype denoted by the <type identifier> does not permit null values, then the DATA_EXCEPTION exception is raised.

13) The result of a nonnull CONVERT_TO is the value of its <value specification>, typed according to the <type identifier>. If the result of the <value specification> does not belong to the subtype denoted by the <type identifier>, then the DATA_EXCEPTION exception is raised.

Case:

a) Conversion of an integer value to an integer value shall be exact.

b) Conversion of an integer or a floating point value to a floating point value shall retain at least the accuracy of the subtype denoted by the <type identifier>.

c) Conversion of a character string value to a character string value shall be as follows:

i) The result character string shall have as many characters as the <value specification> character string.

Case:

1) If the subtype denoted by the <type identifier> is an unconstrained character string, then the maximum number of characters in a string of that subtype shall not be less than the number of characters in the character string <value specification>, otherwise the DATA_EXCEPTION exception is raised.

2) If the subtype denoted by the <type identifier> is a constrained character string, then the number of characters in the character string subtype shall be the same as the number of characters in the character string <value specification>, otherwise the DATA_EXCEPTION exception is raised.

ii) Successive characters in the result are set to successive characters in the <value specification>, converted to the component data type of the character string data type denoted by the <type identifier>. If any character in the <value specification> does not belong to the component subtype, then the DATA_EXCEPTION exception is raised.

d) Conversion of an enumeration value to another enumeration value shall be according to matching enumeration literals.

14) A value to be assigned to an <out variable> shall belong to the data type of that <out variable>; otherwise, the DATA_EXCEPTION exception is raised. If this rule indicates that the

DATA_EXCEPTION exception is to be raised, and the rules of 8.6, <fetch statement> or 8.10, <select statement> indicate that the CONSTRAINT_ERROR exception is to be raised for the same assignment, then the DATA_EXCEPTION exception is the one that is raised.

15) Assignment of a value to the variable denoted by the <variable name> of a <out variable> that is of the form <type mark> (<variable name>) converts the data retrieved from the database from the data type denoted by the <type mark> to the data type of the variable.

a) Conversion of an integer value to an integer value shall be exact.

b) Conversion of a floating point value to an integer value shall round to the nearest integer; a result halfway between two integers (to the accuracy of the floating point data type) may be rounded either up or down. Any program whose effect depends on the direction of rounding of values halfway between two integers is erroneous.

c) Conversion of an integer or a floating point value to a floating point value shall retain at least the accuracy of the subtype of the variable.

d) Conversion of character string values is implicit in the character-by-character assignment described for them. However, the following conditions shall hold; otherwise, the CONSTRAINT_ERROR exception is raised.

i) The data type of the variable and that denoted by the <type mark> shall both have component subtypes allowing the same range of characters.

ii) If the <type mark> denotes an unconstrained character string subtype then the index bounds of the variable shall belong to the index subtype of the unconstrained character string subtype.

iii) If the <type mark> denotes a constrained character string subtype, then the number of characters in the denoted subtype shall be the same as the number of characters in the variable.

iv) If this rule indicates that the CONSTRAINT_ERROR exception is to be raised, and General Rule 14, or the rules of 8.6, <fetch statement> or 8.10, <select statement> indicate that the DATA_EXCEPTION exception is to be raised for the same assignment, then the CONSTRAINT_ERROR exception is the one that is raised.

e) Conversion of an enumeration value to another enumeration value shall be according to matching enumeration literals.

16) If the value to be assigned to an <indicator variable> does not belong to the subtype of its contained <variable name>, then the CONSTRAINT_ERROR exception is raised.

NOTE: Additional rules relevant to <target specification>s may be found in 8.6, <fetch statement>, and 8.10, <select statement>.

Notes

1) The functions effectively declared for <value specification> have two classes of return type, based on the context in which the <value specification> appears, as follows (note that <value specification>s not containing the <key word> USER or an <indicator specification> are Ada program expressions – no effective Ada/SQL functions are declared for them):

VALUE_EXPRESSION class - used in contexts where a <value specification> is used as a <value expression>

VALUE_SPECIFICATION class - used in contexts where the syntax specifically requires a <value specification>, other than within <value expression>: <in value list>, <like predicate>, <insert value>

The VALUE_EXPRESSION class contains four subclasses of return type, based on the context in which the <value expression> containing the <value specification> appears. The subclass names and the contexts are the same as the class names and contexts for <value expression>, as described in Note 1 of section 5.9. Lists of contexts in which each subclass of return type is effectively used for a <value expression> that is a <value specification> can therefore be found in that Note; only brief descriptions of those contexts are given here. The four subclasses of return type, and the contexts in which they are used, are:

VALUE_EXPRESSION - used in contexts where the data type of the <value specification> is not important for the effective Ada declarations.

VALUE_EXPRESSION_ct (typed according to program type) - used in contexts where the result of the <value specification> will be used in an operation for which the effective Ada declarations are defined with strongly-typed operands.

VALUE_EXPRESSION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used in contexts where the result of the <value specification> is the operand of a CONVERT_TO.

VALUE_EXPRESSION_y, where y is INTEGER or FLOATING - used in contexts where the result of the <value specification> will be used in an operation with only one operand, where the type class (integer or floating point vs. character string and enumeration) of the operand is important, and where the context of the operation is such that the function effectively declared for it returns a result of type VALUE_EXPRESSION or VALUE_EXPRESSION_y (i.e., not strongly typed and not CONVERT_TO). Note that two of the same types (VALUE_EXPRESSION_INTEGER and VALUE_EXPRESSION_FLOATING) are used for both this subclass and the previous subclass, so the effective <value specification> functions with those return types apply to both subclasses. Also note that the VALUE_EXPRESSION_y classs of functions effectively declared for <value expression>, as well as for <column specification>, also return values of type VALUE_EXPRESSION_STRING and VALUE_EXPRESSION_ENUMERATION. There is no context in which a <value specification> may be used that would make these return types applicable to the effective functions declared for <value specification>, however.

The VALUE_SPECIFICATION class contains three subclasses of return type, based on the context in which the <value specification> appears, as follows:

VALUE_SPECIFICATION - used in contexts where the data type of the <value specification> is not important for the effective Ada declarations. Relevant contexts immediately containing <value specification>:

<insert value>

VALUE_SPECIFICATION_ct (typed according to program type) - used in contexts where the result of the <value specification> will be used in an operation for which the effective Ada declarations are defined with strongly-typed operands. Relevant contexts immediately containing <value specification>:

<in value list>

<like predicate>

VALUE_SPECIFICATION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used when the result of the <value specification> is the operand of a CONVERT_TO operation.

Note that no VALUE_SPECIFICATION_y return type class is required, unlike the effective functions declared for <value expression> and <column specification>, which require VALUE_EXPRESSION_y and COLUMN_SPECIFICATION_y return classes, respectively. There is no context in which a <value specification> may be used that would make this class of return type applicable to the effective functions declared for <value specification>.

2) The INDICATOR effective functions are used to flag whether or not a particular program value is null, and they also effectively convert values from their program representation (parameter of program data type ct) to their internal Ada/SQL effective type representation (various return types).

3) Since the value returned by the <key word> USER is taken to be of character string data type DATABASE.USER_AUTHORIZATION_IDENTIFIER, the appropriate USER functions are effectively declared returning the six relevant Ada/SQL effective type representations (replace x with either VALUE_SPECIFICATION or VALUE_EXPRESSION): x_DATABASE_USER_AUTHORIZATION_IDENTIFIER (for strongly typed contexts), x_STRING (for contexts in which the type class is important), and x (for contexts in which operands to the effective functions are untyped).

4) The effective type representations returned by the effective USER functions just noted cannot be used as parameters to the effective INDICATOR functions described above, since those parameters must be of program types. However, the <key word> USER is permitted to be the operand of an <indicator specification>. To provide effective declarations for this, an effective USER function, returning a value of effective type USER_VALUE_SPECIFICATION, is declared. Six INDICATOR functions, with this type as parameter and the appropriate return types (same as for effective USER functions discussed in previous note) are then effectively declared.

5) There are several CONVERT_TO functions effectively declared for each subtype, ct, to which a value may be converted. These return a value typed in one of the VALUE_SPECIFICATION subclasses, as described in Note 1, above. The <value specification> CONVERT_TO functions are different from those defined for <value expression> in 5.9, since the former are used in contexts

where only a <value specification> is permitted, while the latter are used in contexts where any type of <value expression> is permitted by the BNF. The <value specification> CONVERT_TO functions are, of course, also different from those defined for <column specification> in 5.7, which are used in contexts where only a <column specification> is permitted.

The effective type of the parameter to each of the CONVERT_TO functions denotes the classes of data types that may be converted to the target data type. Thus, only integer values may be converted to an integer data type, either integer or floating point values may be converted to a floating point data type, and only string values may be converted to a string data type. Enumeration data types are themselves divided into classes, since (by Ada rules) a conversion is allowed from an operand type to a target type if one of the two types is derived from the other, directly or indirectly, or if there exists a third type from which both types are derived, directly or indirectly. This enables enumeration types to be partitioned into classes, with types in the same class being mutually convertible while those in different classes are not. Within each class there exists a data type that is not a derived type; all other types in the class are derived, directly or indirectly, from it. We use this so-called ultimate parent type to designate the class.

We have elected to type the parameter of effective CONVERT_TO functions according to the appropriate data type class. It would also be feasible to strongly type the parameter according to the actual data type of the operand being converted. An implementation which generates all possible effective Ada/SQL declarations based on the type declarations contained in a schema would then have to generate order(n**2) functions, however, where n is the number of different data types declared. By using type classes, the number of functions that must be generated is order(n).

6) The Ada/SQL <value specification> conforms to the ANSI SQL <value specification>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	-
SR2	—	7
SR3	SR2	8
SR4	SR3	-
SR5	—	9
SR6	SR4	10
—	SR5	11
—	SR6	12
—	SR7	13
—	SR8-SR10	14
—	SR11	15
—	SR12	16
—	SR13	17
—	SR14	18
—	SR15	19
—	SR16	20
—	SR17	21
—	SR18	22
—	SR19	23
—	SR20	24
—	SR21	25
—	SR22	26
—	SR23	27
—	SR24-SR25	28
—	SR26	29
—	SR27	30
—	SR28	31
—	GR1	-
—	GR2-GR3	32
—	GR4	33
—	GR5	34
—	GR6	35
GR1	—	36
GR2	GR7	37
GR3	GR8	-
GR4	GR9	38
—	GR10	39
—	GR11	40
—	GR12-GR13	41
—	GR14	42
—	GR15	43
—	GR16	44

7) ANSI SQL <parameter specification>s are not relevant to Ada/SQL.

8) Ada/SQL uses Ada's strong typing to define indicator values. INDICATOR_VARIABLE is an enumeration type with values NOT_NULL and NULL_VALUE.

9) ANSI SQL <parameter specification>s are not relevant to Ada/SQL. The restriction on the use of <variable specification> given in ANSI SQL SR5 is not relevant to Ada/SQL, since Ada/SQL is essentially entirely an embedded language, and <variable specification>s can therefore be used

anywhere within it.

10) Ada/SQL SR4 expresses one aspect of Ada/SQL's strong typing.

Release 1 implementations do not support the <key word> USER.

11) An ANSI SQL <value specification> is limited to one program value and (optionally) one indicator, with no operators permitted. Ada/SQL extends this slightly, by allowing arithmetic expressions and various data typing operations within <value specification>s. Ada/SQL does enforce the ANSI SQL restriction of only one indicator within a <value specification>, however.

12) Ada/SQL SR6 expresses one aspect of Ada/SQL's strong typing.

13) A <value specification> not contained in one of the contexts mentioned in SR7 must, by virtue of the grammar and other Syntax Rules, be immediately contained in a <value expression>. Arithmetic operators may be applied to <value specification>s contained within <value expression>s by using the <value expression> arithmetic operator syntax, which happens to be the same as that defined for <value specification>. Not allowing <value specification> arithmetic operators to appear immediately within <value expression>s is merely a syntactic device to avoid parsing ambiguity; the language would look the same whether they were permitted or not.

It should be noted that the arithmetic operators allowed in a <value specification> are those of ANSI SQL; Ada defines several more. It would be possible to implement an Ada/SQL system that allowed all Ada arithmetic operators in <value specification>s, since <value specification>s (other than the <key word> USER) represent program values that can be computed independently of the database. Ada/SQL <value specification> operators are restricted to those of ANSI SQL because it was felt that allowing extra operators in some portions of the language but not in others would be confusing to users.

14) Ada/SQL SR8-SR10 express various aspects of Ada/SQL's strong typing. SR9 embodies Ada rules for handling numbers of type universal_integer and universal_real.

15) The effective Ada declarations for <indicator specification> and CONVERT_TO are functions that return values of special internal Ada/SQL types. <indicator specification>s are designed to be used with user-defined (or predefined) program types, and so cannot be used on the result of an <indicator specification> or a CONVERT_TO.

16) Ada/SQL SR12 ensures that the type of the <value specification> operand to an effective INDICATOR function is known from the source text. This is to avoid ambiguities in the effective Ada functions. For example, consider two floating point data types, ANNUAL_SALARY and MONTHLY_SALARY, two program variables, ANNUAL_PAY and MONTHLY_PAY, of those two data types, respectively, and a program containing the following two fragments of <insert statement>s (PAY_KNOWN is a variable of type INDICATOR_VARIABLE):

```
. . . VALUES <= INDICATOR ( ANNUAL_PAY , PAY_KNOWN ) and . . .

. . . VALUES <= INDICATOR ( MONTHLY_PAY , PAY_KNOWN ) and . . .
```

An implementation which actually generates the effective Ada declarations must generate at least the following:

```
function INDICATOR
         ( VALUE     : ANNUAL_SALARY;
           INDICATOR : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION;

function INDICATOR
         ( VALUE     : MONTHLY_SALARY;
           INDICATOR : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_SPECIFICATION;
```

Now, suppose the same program also contains the following fragment of an <insert statement>, in violation of SR12:

```
. . . VALUES <= INDICATOR ( 25_000.00 , PAY_KNOWN ) and . . .
```

25_000.00 is a literal value of both ANNUAL_SALARY and MONTHLY_SALARY data types, so that the effective INDICATOR function is ambiguous. To comply with SR12, the type of the literal can be explicitly qualified, as in the following legal fragment:

```
. . . VALUES <= INDICATOR ( ANNUAL_SALARY' ( 25_000.00 ) , PAY_KNOWN )
and . . .
```

Release 1 implementations do not enforce SR12 if the <value specification> is of an integer, floating point, or character string data type. Instead, the type of the <value specification> is assumed to be STANDARD.INTEGER, STANDARD.FLOAT, or STANDARD.STRING, as appropriate.

17) Release 1 implementations do not support <indicator value>s. Hence, the optional <indicator value> cannot be used in an <indicator specification>.

18) Ada/SQL SR14 expresses an Ada rule with respect to type conversions and qualifications.

19) The effective Ada declarations for <indicator specification> and the <key word> USER are functions that return values of special internal Ada/SQL types. Ada/SQL <Ada type qualification>s and <Ada type conversion>s are designed to be used with user-defined (or predefined) program types, and so cannot be used on the result of an <indicator specification> or the <key word> USER.

20) Ada/SQL SR16 ensures that the type of the operand of an <Ada type conversion> is known from the source text, in accordance with the Ada type conversion requirement that "the type of the operand of a type conversion must be determinable independently of the context."

21) Ada/SQL SR17 restates an Ada requirement for qualified expressions.

22) The rules for <Ada type conversion>s differ from those for CONVERT_TO. CONVERT_TO is provided for <value specification>s in order to provide syntax that is consistent with that of <column specification> and <value expression>. <Ada type conversion>s are provided because

they are an integral part of the Ada language. Thus, even though both forms of type conversion differ, each may find a useful place in an Ada/SQL program.

23) Based on the SDL syntax rules, the expanded name of a user-defined subtype denoted by a <type identifier> is <library package name>.ADA_SQL.<type identifier>. The designation of a CONVERT_TO for that subtype is, however, CONVERT_TO.<library package name>.<type identifier>, omitting the ADA_SQL. Omitting the ADA_SQL does not introduce any ambiguities, because the only <type identifier>s contained in the named library package that may be referenced in a CONVERT_TO are those defined in the ADA_SQL nested package.

24) A <value specification> not containing an <indicator specification> or the <key word> USER is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If CONVERT_TO were allowed to data type t1, defined in library package p, from such a <value specification> of data type t2, then the following CONVERT_TO function (as well as others) must be effectively declared:

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function t1 ( LEFT : t2 ) return VALUE_SPECIFICATION_t1;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

An implementation which generates all possible effective Ada/SQL declarations, based on the type declarations contained in a schema, would have to generate order(n**2) such functions, where n is the number of different data types declared. By prohibiting the <value specification> operand of a CONVERT_TO from being of a program type, SR20 ensures that the number of functions effectively declared is linear in the number of user data types defined, rather than depending on the square of the number of data types. Note that the effective Ada/SQL declarations are such that a <value specification> containing an <indicator specification> or the <key word> USER, when used as the operand to a <value specification> CONVERT_TO, effectively returns an object of one of the VALUE_SPECIFICATION_x types described in Note 1, rather than a type unique to its underlying conceptual type, thereby avoiding the n**2 problem.

The restriction of Ada/SQL SR20 should actually have virtually no impact on Ada/SQL programmers, because <Ada type conversion>s (described in this section) can be used on the prohibited <value specification>s. CONVERT_TO does provide more flexibility on character string conversions than does Ada type conversion (see Note 41). To use this flexibility on program objects, the program object can be expressed as an <indicator specification> to comply with Ada/SQL SR20. For example, the following violates SR20 (assume that SOCIAL_SECURITY_NUMBER is a program variable of a character string type, with components of a data type other than those of character string type IDENTIFICATION_NUMBER, defined in package P):

```
. . . CONVERT_TO.P.IDENTIFICATION_NUMBER
( SOCIAL_SECURITY_NUMBER ) . . .
```

However, the following has the exact same effect, and is permitted:

```
. . . CONVERT_TO.P.IDENTIFICATION_NUMBER
( INDICATOR ( SOCIAL_SECURITY_NUMBER ) ) . . .
```

25) The only context not mentioned in SR21 in which a <value specification> can appear (other than in another <value specification>) is <value expression>. Type conversions may be applied to <value specification>s contained within <value expression>s by using the CONVERT_TO syntax defined for <value expression>, which happens to be the same as that defined for <value specification>. Not allowing <value specification> CONVERT_TOs to appear within <value expression>s is merely a syntactic device to avoid parsing ambiguity; the language would look the same whether it were permitted or not.

26) CONVERT_TO is designed such that only types of the same class are mutually convertible: numerics to numerics, character strings to character strings, and enumerations to enumerations within the same ultimate parent type. Numerics are not totally mutually convertible, however. Although both integer and floating point values can be converted to a floating point type, only integer values can be converted to an integer type. Why not permit a floating point value to be converted to an integer type, particularly since "all numbers are comparable" in ANSI SQL? The ANSI comparability statement notwithstanding, ANSI SQL does not permit approximate numeric values (the analog of Ada/SQL floating point) to be assigned to exact numeric (a superset of Ada/SQL integers) database columns or program variables. To enforce this restriction in Ada/SQL, CONVERT_TO is not allowed from a floating point value to an integer type. The net result is that any database computations involving both integer and floating point values must be done in a floating point type. This would most likely be the desired mode anyway, due to the possible loss of precision when using integer arithmetic.

    Note that the CONVERT_TO prohibition on converting floating point to integer is in contrast to the <Ada type conversion>, in which floating point values may be converted to an integer type. Even though the CONVERT_TO discussed here operates on <value specification>s, which are program values rather than database values, it was felt best to establish the same restrictions on <value specification> CONVERT_TOs as on those for <column specification>s and <value expression>s.

27) The <program variable> of a <target specification> specifies the program variable to which a database value is to be assigned, and the <target specification> is considered typed according to its <program variable> in order to implement the relevant aspect of Ada/SQL's strong typing.

28) Ada/SQL SR24 and SR25 express two aspects of Ada/SQL's strong typing.

    Release 1 implementations do not support <indicator variable>s. Hence, the optional <indicator variable> may not be used in a <target specification>.

29) The declaration of an enumeration or unconstrained character string data type declares a <type mark> denoting the base type. The declaration of an integer, floating point, or constrained character string data type, as well as the declaration of a derived data type, declares an anonymous base type with the <type mark> denoting a subtype (the first-named subtype). Each parameter to an INTO procedure of the effective Ada declarations is declared using a <type mark> that denotes a base type or a first-named subtype, which we have called a "data type". Parameters are not declared for every subtype, since the parameter and result type profile of a subprogram considers only base type, not subtype. According to Ada rules, the <type mark> of an <out variable> actual

parameter to an INTO procedure must conform to the type mark used in declaring the corresponding formal parameter of that procedure. This is the reason that SR26 requires that the <type mark> denote a data type (base type or first-named subtype).

30) The allowed conversions for an <out variable> containing a <type mark> are in accordance with Ada type conversion rules. These are different from those of the Ada/SQL CONVERT_TO operator.

31) The typing rules given in SR28 for an <out variable> containing a <type mark> are consistent with that construct's interpretation in terms of the effective Ada declarations: a parameter, in the form of an Ada type conversion, to an effective INTO procedure.

32) The <value expression> division operator raises the DATA_EXCEPTION exception for division by 0, in contrast to the NUMERIC_ERROR exception required by GR3. Since <value specification>s contain program values, the standard Ada exception, NUMERIC_ERROR, is used for them, rather than DATA_EXCEPTION, which is used for database errors. Also note that the requirement, of both GR2 and GR3, that the result of an arithmetic operation belong to the Ada base type of the operands (except for universal data types), is based on Ada arithmetic semantics. It is highly desirable that an implementation raise NUMERIC_ERROR on computations out of range, but Ada does not require this, recognizing that detecting overflow may be difficult in some environments.

33) Arithmetic operators within <value specification>s, which may contain only program values, would be executed in runtime systems as the standard Ada predefined functions. Consequently, GR4 expresses the Ada rules for expression evaluation. Note that these are different from the Ada/SQL rules given for <value expression>, which may contain database values, and so must be evaluated by the database management system.

34) Ada/SQL GR5 describes the Ada order of expression evaluation, which is not the same as that of ANSI SQL. The properties of the arithmetic operators are such, however, that results should be the same regardless of which order of evaluation is used.

35) Ada/SQL GR6 is similar to the Ada rules for using variables with undefined values.

36) ANSI SQL <parameter specification>s are not relevant to Ada/SQL.

37) An <indicator specification> need not contain an <indicator value>; this is to allow simple use of the INDICATOR syntax where required for adherence to other Syntax Rules (see, for example, Note 24). An <indicator specification> without an <indicator value> indicates a non-null value, so that INDICATOR (V), where V is a <value specification>, is equivalent to V.

38) In ANSI SQL, the value specified by USER is the <authorization identifier> of the <module> that is associated with the executing program. This would presumably not change across executions of the program. To be useful, USER should really specify a value that is indicative of the individual causing the program to be executed. This is, in fact, what virtually all database management systems have implemented, and is the subject of a planned revision to the ANSI SQL specification. Ada/SQL leaves the assignment of <authorization identifier> to program execution as

implementor-defined, so that an Ada/SQL implementation is free to assign a non-varying <authorization identifier> to a program, similar to ANSI SQL, or to return a value truly indicative of the user.

39) Ada/SQL GR10 expresses the Ada rules for type qualification, as they are relevant to the data types supported by Ada/SQL. In a runtime implementation, Ada/SQL <Ada type qualification>s would be actually executed as Ada type qualifications.

40) Ada/SQL GR11 expresses the Ada rules for type conversion, as they are relevant to the data types supported by Ada/SQL. In a runtime implementation, Ada/SQL <Ada type conversion>s would be actually executed as Ada type conversions. Note that the Ada rules for type conversion differ from the Ada/SQL rules described for the CONVERT_TO syntax.

41) The General Rules for <value specification> CONVERT_TO differ from those for <column specification> and <value expression> CONVERT_TO in that the former requires raising the DATA_EXCEPTION exception for violations of subtype constraints, whereas the latter merely states that programs causing subtype constraints to be violated are erroneous. The latter CONVERT_TOs are performed on database values, where subtype constraints are difficult to check if the database management system does not support such checking. The former CONVERT_TOs, on the other hand, are performed on program values, where subtype checking can readily be accomplished.

    Release 1 implementations do not check for CONVERT_TO subtype constraint violations; the DATA_EXCEPTION exception is not raised.

    Note that Ada/SQL character string conversion includes a type conversion for each character in the string. This is in contrast to Ada type conversion for strings (including that discussed for <Ada type conversion>, above), which requires that two string types have the same component type in order to be mutually convertible. The extended Ada/SQL convertibility is provided to match the functionality of ANSI SQL, in which all character strings are comparable.

42) The data type of an <out variable> must be the same as that of the corresponding database column from which data is being retrieved. It is possible, however, that values stored in a database not supporting subtype checking may lie outside the range of the appropriate subtype. Although this error may not be readily detectable when the bogus data are created (if done entirely within the database), it is easily detected when the data are retrieved. In an implementation that actually executes the effective Ada declarations, the CONSTRAINT_ERROR exceptions noted in 8.6 and 8.10 involve subtype checking on subprogram **out** parameters. This CONSTRAINT_ERROR checking would be performed on subprogram return, whereas the DATA_EXCEPTION checking noted in GR14 would be performed in the body of the subprogram. Hence, the DATA_EXCEPTION exception takes precedence over the CONSTRAINT_ERROR exception.

43) The requirements expressed in GR15d correspond to the runtime checks performed on Ada array type conversions, as applicable to character string <out variable>s expressed as Ada type conversions. In an implementation which actually executed the effective Ada declarations, these checks would be performed at the point of calling an INTO procedure. Failure of any check would raise CONSTRAINT_ERROR, without actually calling the procedure. This is the reason that CONSTRAINT_ERROR here has precedence over DATA_EXCEPTION elsewhere.

44) Although an <indicator variable> must be of data type INDICATOR_VARIABLE, it is possible for the variable denoted by the <variable name> to belong to a user-defined subtype that does not include the value to be assigned to the <indicator variable>.

5.7 <column specification>

Function

Reference a named column.

Format

<column specification> ::=
  <column specification type conversion>
 | [<qualifier> .] <column name>

<column specification type conversion> ::=
  CONVERT_TO . <library package name> . <type identifier>
  (<column specification>)

<qualifier> ::=
  <table name> | <correlation name>

Effective Ada Declarations

```
type COLUMN_SPECIFICATION is private;

type COLUMN_SPECIFICATION_INTEGER is private;

type COLUMN_SPECIFICATION_FLOATING is private;

type COLUMN_SPECIFICATION_STRING is private;

type COLUMN_SPECIFICATION_ENUMERATION is private;
```

For a program data type ct:

```
type COLUMN_SPECIFICATION_ct is private;
```

For an enumeration data type ct that is not derived from another enumeration type (an ultimate parent type):

```
type COLUMN_SPECIFICATION_ENUMERATION_ct is private;
```

For a column with <column name> c, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION;

function c return VALUE_EXPRESSION;
```

```
function c return GROUP_BY_CLAUSE;
```

For a column with <column name> c, of an integer type, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one such column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION_INTEGER;

function c return VALUE_EXPRESSION_INTEGER;
```

For a column with <column name> c, of a floating point type, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one such column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION_FLOATING;

function c return VALUE_EXPRESSION_FLOATING;
```

For a column with <column name> c, of a character string type, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one such column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION_STRING;

function c return VALUE_EXPRESSION_STRING;
```

For a column with <column name> c, of an enumeration type with ultimate parent type ct, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one such column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION_ENUMERATION;

function c return COLUMN_SPECIFICATION_ENUMERATION_ct;

function c return VALUE_EXPRESSION_ENUMERATION;

function c return VALUE_EXPRESSION_ENUMERATION_ct;
```

For a column with <column name> c, of data type ct, declared in table t with <authorization identifier> a, the following functions are effectively declared (1) for references as a <column specification> containing only the <column name> (there may be more than one such column named "c"; the following functions are effectively declared only once for all of them for this purpose), (2) in generic package t_CORRELATION.NAME, and (3) in generic package a_t_CORRELATION.NAME:

```
function c return COLUMN_SPECIFICATION_ct;

function c return VALUE_EXPRESSION_ct;
```

For a column with <column name> c, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_UNTYPED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_UNTYPED_a_t;

type VALUE_EXPRESSION_UNTYPED_a_t is
  record
    . . .
    c : VALUE_EXPRESSION;
    . . .
  end record;

function t return VALUE_EXPRESSION_UNTYPED_a_t;

type GROUP_BY_CLAUSE_a_t is
  record
    . . .
    c : GROUP_BY_CLAUSE;
    . . .
  end record;

function t return GROUP_BY_CLAUSE_a_t;
```

For a column with <column name> c, of an integer type, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_INTEGER;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_CLASSED_a_t;

type VALUE_EXPRESSION_CLASSED_a_t is
  record
    . . .
    c : VALUE_EXPRESSION_INTEGER;
    . . .
  end record;
```

```
function t return VALUE_EXPRESSION_CLASSED_a_t;
```

For a column with <column name> c, of a floating point type, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_FLOATING;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_CLASSED_a_t;

type VALUE_EXPRESSION_CLASSED_a_t is
  record
    . . .
    c : VALUE_EXPRESSION_FLOATING;
    . . .
  end record;

function t return VALUE_EXPRESSION_CLASSED_a_t;
```

For a column with <column name> c, of a character string type, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_STRING;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_CLASSED_a_t;

type VALUE_EXPRESSION_CLASSED_a_t is
  record
    . . .
    c : VALUE_EXPRESSION_STRING;
    . . .
  end record;

function t return VALUE_EXPRESSION_CLASSED_a_t;
```

For a column with <column name> c, of an enumeration type, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_ENUMERATION;
```

```
    . . .
  end record;

function t return COLUMN_SPECIFICATION_CLASSED_a_t;

type VALUE_EXPRESSION_CLASSED_a_t is
  record
    . . .
    c : VALUE_EXPRESSION_ENUMERATION;
    . . .
  end record;

function t return VALUE_EXPRESSION_CLASSED_a_t;
```

For a column with <column name> c, of an enumeration type with ultimate parent type ct, in table t with <authorization identifier> a (all other <column name>s of columns of enumeration types within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_ENUMERATION_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_ENUMERATION_ct;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_CLASSED_ENUMERATION_a_t;

type VALUE_EXPRESSION_CLASSED_ENUMERATION_a_t is
  record
    . . .
    c : VALUE_EXPRESSION_ENUMERATION_ct;
    . . .
  end record;

function t return VALUE_EXPRESSION_CLASSED_ENUMERATION_a_t;
```

For a column with <column name> c, of data type ct, in table t with <authorization identifier> a (all other <column name>s within a.t are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_TYPED_a_t is
  record
    . . .
    c : COLUMN_SPECIFICATION_ct;
    . . .
  end record;

function t return COLUMN_SPECIFICATION_TYPED_a_t;

type VALUE_EXPRESSION_TYPED_a_t is
  record
    . . .
```

```
    c : VALUE_EXPRESSION_ct;
    . . .
  end record;

function t return VALUE_EXPRESSION_TYPED_a_t;
```

For a table t with <authorization identifier> a (components for all other other tables with <authorization identifier> a are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_UNTYPED_a is
  record
    . . .
    t : COLUMN_SPECIFICATION_UNTYPED_a_t;
    . . .
  end record;

function a return COLUMN_SPECIFICATION_UNTYPED_a;

type VALUE_EXPRESSION_UNTYPED_a is
  record
    . . .
    t : VALUE_EXPRESSION_UNTYPED_a_t;
    . . .
  end record;

function a return VALUE_EXPRESSION_UNTYPED_a;

type GROUP_BY_CLAUSE_a is
  record
    . . .
    t : GROUP_BY_CLAUSE_a_t;
    . . .
  end record;

function a return GROUP_BY_CLAUSE_a;

type COLUMN_SPECIFICATION_CLASSED_a is
  record
    . . .
    t : COLUMN_SPECIFICATION_CLASSED_a_t;
    . . .
  end record;

function a return COLUMN_SPECIFICATION_CLASSED_a;

type VALUE_EXPRESSION_CLASSED_a is
  record
    . . .
    t : VALUE_EXPRESSION_CLASSED_a_t;
    . . .
  end record;
```

```
function a return VALUE_EXPRESSION_CLASSED_a;

type COLUMN_SPECIFICATION_TYPED_a is
  record
    . . .
    t : COLUMN_SPECIFICATION_TYPED_a_t;
    . . .
  end record;

function a return COLUMN_SPECIFICATION_TYPED_a;

type VALUE_EXPRESSION_TYPED_a is
  record
    . . .
    t : VALUE_EXPRESSION_TYPED_a_t;
    . . .
  end record;

function a return VALUE_EXPRESSION_TYPED_a;
```

For a table t with at least one column of an enumeration type, with <authorization identifier> a (components for all other tables with at least one column of an enumeration type and <authorization identifier> a are similarly included in the record type declarations):

```
type COLUMN_SPECIFICATION_CLASSED_ENUMERATION_a is
  record
    . . .
    t : COLUMN_SPECIFICATION_CLASSED_ENUMERATION_a_t;
    . . .
  end record;

function a return COLUMN_SPECIFICATION_CLASSED_ENUMERATION_a;

type VALUE_EXPRESSION_CLASSED_ENUMERATION_a is
  record
    . . .
    t : VALUE_EXPRESSION_CLASSED_ENUMERATION_a_t;
    . . .
  end record;

function a return VALUE_EXPRESSION_CLASSED_ENUMERATION_a;
```

For an integer program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
 ...
 package p is
  ...
  function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
  return COLUMN_SPECIFICATION_dt;

  function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
```

```
    return COLUMN_SPECIFICATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
     return COLUMN_SPECIFICATION_INTEGER;
    ...
  end p;
  ...
end CONVERT_TO;
```

For a floating point program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : COLUMN_SPECIFICATION_FLOATING )
     return COLUMN_SPECIFICATION_dt;

    function ct ( LEFT : COLUMN_SPECIFICATION_FLOATING )
     return COLUMN_SPECIFICATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_FLOATING )
     return COLUMN_SPECIFICATION_FLOATING;

    function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
     return COLUMN_SPECIFICATION_dt;

    function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
     return COLUMN_SPECIFICATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_INTEGER )
     return COLUMN_SPECIFICATION_FLOATING;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

For a character string program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : COLUMN_SPECIFICATION_STRING )
     return COLUMN_SPECIFICATION_dt;

    function ct ( LEFT : COLUMN_SPECIFICATION_STRING )
     return COLUMN_SPECIFICATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_STRING )
```

```
      return COLUMN_SPECIFICATION_STRING;
      . . .
    end p;
    . . .
end CONVERT_TO;
```

For an enumeration program subtype ct defined in library package p, of data type dt with ultimate parent type pt (ct may be the same as dt, and dt may be the same as pt):

```
package CONVERT_TO is
   . . .
  package p is
    . . .
    function ct ( LEFT : COLUMN_SPECIFICATION_ENUMERATION_pt )
     return COLUMN_SPECIFICATION_dt;

    function ct ( LEFT : COLUMN_SPECIFICATION_ENUMERATION_pt )
     return COLUMN_SPECIFICATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_ENUMERATION_pt )
     return COLUMN_SPECIFICATION_ENUMERATION;

    function ct ( LEFT : COLUMN_SPECIFICATION_ENUMERATION_pt )
     return COLUMN_SPECIFICATION_ENUMERATION_pt;
    . . .
  end p;
  . . .
end CONVERT_TO;
```

Example

```
CURSOR      : CURSOR_NAME;
BOSSES_NAME : EXAMPLE_TYPES.ADA_SQL.BOSS_NAME;

package E is new EMPLOYEE_CORRELATION.NAME ( "E" ); -- employees
package M is new EMPLOYEE_CORRELATION.NAME ( "M" ); -- managers
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => SALARY > 25_000.00 ) ); -- variation: EMPLOYEE.SALARY
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( E.NAME & E.SALARY & M.NAME & M.SALARY,
  FROM   => E.EMPLOYEE & M.EMPLOYEE,
  WHERE  => EQ ( E.MANAGER , M.NAME )
  AND       E.SALARY > M.SALARY ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( NAME & SALARY,
  FROM   => EMPLOYEE,
  WHERE  => LIKE
```

```
( CONVERT_TO.EXAMPLE_TYPES.BOSS_NAME ( NAME ), BOSSES_NAME ) ) );
```

Syntax Rules

1) A <column specification> references a named column. The meaning of a reference to a column depends on the context.

2) Let C be the <column name> of the <column specification>.

3) Case:

   a) If a <column specification> contains a <qualifier>, then the <column specification> shall appear within the scope of one or more <table name>s or <correlation name>s equal to that <qualifier>. If there is more than one such <table name> or <correlation name>, then the one with the most local scope is specified. The table associated with the specified <table name> or <correlation name> shall include a column whose <column name> is C.

   b) If a <column specification> does not include a <qualifier>, then it shall be contained within the scope of one or more <table name>s or <correlation name>s. Of these, let the phrase "possible qualifiers" denote those <table name>s and <correlation name>s whose associated table includes a column whose <column name> is C. There shall be exactly one possible qualifier with the most local scope, and that <table name> or <correlation name> is implicitly specified.

   NOTE: The "scope" of a <table name> or <correlation name> is specified in 5.20, "<from clause>", 8.5, "<delete statement: searched>", 8.11, "<update statement: positioned>", and 8.12, "<update statement: searched>".

4) If a <column specification> is contained in a <table expression> T and the scope of the implicitly or explicitly specified <qualifier> of the <column specification> is some <SQL statement> or <table expression> that contains the <table expression> T, then the <column specification> is an "outer reference" to the table associated with that <qualifier>.

5) A <column specification type conversion> shall only appear within a <distinct set function> or a <like predicate>.

6) Let T denote the table associated with the explicitly or implicitly specified <qualifier> R. The data type of a <column specification> not containing a <column specification type conversion> is the data type of column C of T.

7) The data type of a <column specification type conversion> is that denoted by the <type identifier>, and shall be an integer, floating point, character string, or enumeration data type.

   Case:

   a) If the <library package name> is STANDARD, then the <type identifier> shall be declared within the STANDARD Ada/SQL predefined environment.

b) If the library package denoted by the <library package name> is part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within that library package.

c) If the library package denoted by the <library package name> is not part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within the ADA_SQL nested package of that library package.

8) Case:

a) If the <type identifier> of a <column specification type conversion> denotes an integer data type, then the <column specification> of that <column specification type conversion> shall be of an integer data type.

b) If the <type identifier> of a <column specification type conversion> denotes a floating point data type, then the <column specification> of that <column specification type conversion> shall be of an integer or a floating point data type.

c) If the <type identifier> of a <column specification type conversion> denotes a character string data type, then the <column specification> of that <column specification type conversion> shall be of a character string data type.

d) If the <type identifier> of a <column specification type conversion> denotes an enumeration data type, then the <column specification> of that <column specification type conversion> shall be of an enumeration data type such that both data types have the same ultimate parent type.

9) If a <column specification> contains a <qualifier> that is a <table name> containing only a <table identifier>, then there shall be no other table with the same <table name> declared within any <schema package> named in the program's <context clause> that contains a column with <column name> C.

10) If a <column specification> contains only a <column name>, then all columns with that <column name> C declared within any <schema package> named in the program's <context clause> shall be of the same data type.

General Rules

1) "C" or "R.C" references column C in a given row of T.

2) The result of a <column specification type conversion> with an argument evaluating to the null value is the null value. If the subtype denoted by the <type identifier> does not permit null values, then the program executing the <column specification type conversion> is erroneous.

3) The result of a nonnull <column specification type conversion> is the value of its <column specification> operand, typed according to the <type identifier>. If the value of the <column

specification> does not belong to the subtype denoted by the <type identifier>, then the program executing the <column specification type conversion> is erroneous.

Case:

a) Conversion of an integer value to an integer value shall be exact.

b) Conversion of an integer or a floating point value to a floating point value shall retain at least the accuracy of the subtype denoted by the <type identifier>.

c) Conversion of a character string value to a character string value shall be as follows:

   i) The result character string shall have as many characters as the <column specification> character string.

      Case:

      1) If the subtype denoted by the <type identifier> is an unconstrained character string, then the maximum number of characters in a string of that subtype shall not be less than the number of characters in the character string <column specification>, otherwise the program executing the <column specification type conversion> is erroneous.

      2) If the subtype denoted by the <type identifier> is a constrained character string, then the number of characters in that string subtype shall be the same as the number of characters in the character string <column specification>, otherwise the program executing the <column specification type conversion> is erroneous.

   ii) Successive characters in the result are set to successive characters in the <column specification>, converted to the component data type of the character string data type denoted by the <type identifier>. If any character in the <column specification> does not belong to the component subtype, then the program executing the <column specification type conversion> is erroneous.

d) Conversion of an enumeration value to another enumeration value shall be according to matching enumeration literals.

Notes

1) The functions effectively declared for <column specification> have three classes of return type, based on the context in which the <column specification> appears, as follows:

   GROUP_BY_CLAUSE class - used in <group by clause>s

   VALUE_EXPRESSION class - used in contexts where a <column specification> is used as a <value expression>

COLUMN_SPECIFICATION class - used in contexts where the syntax explicitly requires a <column specification>, other than within <value expression> and <group by clause>: <column specification type conversion>, <distinct set function>, <like predicate>, <null predicate>, <sort specification>

The GROUP_BY_CLAUSE class contains only one type, GROUP_BY_CLAUSE.

The VALUE_EXPRESSION class contains four subclasses of return type, based on the context in which the <value expression> containing the <column specification> appears. The subclass names and the contexts are the same as the class names and contexts for <value expression>, as described in Note 1 of section 5.9. Lists of contexts in which each subclass of return type is effectively used for a <value expression> that is a <column specification> can therefore be found in that Note; only brief descriptions of those contexts are given here. The four subclasses of return type, and the contexts in which they are used, are:

VALUE_EXPRESSION - used in contexts where the data type of the <column specification> is not important for the effective Ada declarations.

VALUE_EXPRESSION_ct (typed according to program type) - used in contexts where the result of the <column specification> will be used in an operation for which the effective Ada declarations are defined with strongly-typed operands.

VALUE_EXPRESSION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used in contexts where the result o. the <column specification> is the operand of a CONVERT_TO.

VALUE_EXPRESSION_y, where y is INTEGER, FLOATING, STRING, or ENUMERATION - used in contexts where the result of the <column specification> will be used in an operation with only one operand, where the type class (integer, floating point, character string, or enumeration) of the operand is important, and where the context of the operation is such that the function effectively declared for it returns a result of type VALUE_EXPRESSION or VALUE_EXPRESSION_y (i.e., not strongly typed and not CONVERT_TO). Note that three of the same types (VALUE_EXPRESSION_INTEGER, VALUE_EXPRESSION_FLOATING, and VALUE_EXPRESSION_STRING) are used for both this subclass and the previous subclass, so the effective <column specification> functions with those return types apply to both subclasses.

The COLUMN_SPECIFICATION class contains four subclasses of return type, based on the context in which the <column specification> appears, as follows:

COLUMN_SPECIFICATION - used in contexts where the data type of the <column specification> is not important for the effective Ada declarations. Relevant contexts immediately containing <column specification>:

COUNT_DISTINCT <distinct set function> (the operand need not be strongly typed; the result is always of type DATABASE.INT)

<null predicate>

<sort specification>

COLUMN_SPECIFICATION_ct (typed according to program type) - used in contexts where the result of the <column specification> will be used in an operation for which the effective Ada declarations are defined with strongly-typed operands. Relevant contexts immediately containing <column specification>:

<distinct set function>, other than COUNT_DISTINCT, when it is in a context requiring strong typing (the function effectively declared for the <distinct set function> returns a result of type VALUE_EXPRESSION_-ct)

<like predicate>

COLUMN_SPECIFICATION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used when the result of the <column specification> is the operand of a <column specification type conversion>.

COLUMN_SPECIFICATION_y, where y is INTEGER, FLOATING, STRING, or ENUMERATION - used when the <column specification> is the operand of a <distinct set function>, other than COUNT_DISTINCT, that is itself used in a context such that the function effectively declared for it returns a result of type VALUE_EXPRESSION or VALUE_EXPRESSION_y (i.e., not strongly typed and not CONVERT_TO). Note that three of the same types (COLUMN_-SPECIFICATION_INTEGER, COLUMN_SPECIFICATION_FLOATING, and COLUMN_-SPECIFICATION_STRING) are used for both this subclass and the previous subclass, so the effective <column specification> functions with those return types apply to both subclasses.

2) The generic packages t_CORRELATION.NAME and a_t_CORRELATION.NAME are used to declare <correlation name>s for table a.t. For example, the <correlation name> cn may be declared as:

package cn **is new** t_CORRELATION.NAME ("cn"); or

package cn **is new** a_t_CORRELATION.NAME ("cn");

The <column specification> cn.c effectively calls one of the c functions declared within the generic package, to return the appropriately typed value representing the <column specification>.

Release 1 implementations do not support <authorization identifier>s within <table name>s. Hence, only generic package t_CORRELATION.NAME is available for table t.

3) The <column specification> t.c effectively calls one of the t functions to return a value of a record type; then selects the c component of this value, which is the appropriately typed value representing the <column specification>.

4) The <column specification> a.t.c effectively calls one of the a functions to return a value of a record type; then selects the t component of this value, which is also of a record type; then selects the c component of that record value, which is the appropriately typed value representing the <column specification>.

Release 1 implementations do not support <authorization identifier>s within <table name>s. Hence, no a functions are effectively declared.

5) There are several CONVERT_TO functions effectively declared for each subtype, ct, to which a value may be converted. These return a value typed in one of the COLUMN_SPECIFICATION subclasses, as described in Note 1, above. The <column specification> CONVERT_TO functions are different from those defined for <value expression> in 5.9, since the former are used in contexts where only a <column specification> is permitted, while the latter are used in contexts where any type of <value expression> is permitted by the BNF.

The effective type of the parameter to each of the CONVERT_TO functions denotes the classes of data types that may be converted to the target data type. Thus, only integer values may be converted to an integer data type, either integer or floating point values may be converted to a floating point data type, and only string values may be converted to a string data type. Enumeration data types are themselves divided into classes, since (by Ada rules) a conversion is allowed from an operand type to a target type if one of the two types is derived from the other, directly or indirectly, or if there exists a third type from which both types are derived, directly or indirectly. This enables enumeration types to be partitioned into classes, with types in the same class being mutually convertible while those in different classes are not. Within each class there exists a data type that is not a derived type; all other types in the class are derived, directly or indirectly, from it. We use this so-called ultimate parent type to designate the class.

We have elected to type the parameter of effective CONVERT_TO functions according to the appropriate data type class. It would also be feasible to strongly type the parameter according to the actual data type of the operand being converted. An implementation which generates all possible effective Ada/SQL declarations based on the type declarations contained in a <schema> would then have to generate order(n**2) functions, however, where n is the number of different data types declared. By using type classes, the number of functions that must be generated is order(n).

6) The Ada/SQL <column specification> conforms to the ANSI SQL <column specification>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1-SR2	SR1-SR2	-
SR3	SR3	7
SR4	SR4	-
—	SR5	8
SR5	SR6	-
—	SR7	9
—	SR8	10
—	SR9	11
—	SR10	12
GR1	GR1	-
—	GR2-GR3	13

7) The last sentence of ANSI SQL SR3b reads "... and that <table name> or <qualifier> is implicitly specified." In Ada/SRL SR3b, <correlation name> is substituted for <qualifier>, since this is really what appears to be intended in the ANSI rule.

8) <column specification>s contained in <group by clause>s, <null predicate>s, and <sort specification>s are considered untyped, and so may not have <column specification type conversion>s applied to them. Type conversions may be applied to <column specification>s contained within <value expression>s by using the CONVERT_TO syntax defined for <value expression>, which happens to be the same as that defined for <column specification type conversion>. Not allowing <column specification type conversion>s to appear within <value expression>s is merely a syntactic device to avoid parsing ambiguity; the language would look the same whether it were permitted or not.

9) Based on the SDL syntax rules, the expanded name of a user-defined subtype denoted by a <type identifier> is <library package name>.ADA_SQL.<type identifier>. The designation of a CONVERT_TO for that subtype is, however, CONVERT_TO.<library package name>.<type identifier>, omitting the ADA_SQL. Omitting the ADA_SQL does not introduce any ambiguities, because the only <type identifier>s contained in the named library package that may be referenced in a CONVERT_TO are those defined in the ADA_SQL nested package.

10) CONVERT_TO is designed such that only types of the same class are mutually convertible: numerics to numerics, character strings to character strings, and enumerations to enumerations with the same ultimate parent type. Numerics are not totally mutually convertible, however. Although both integer and floating point values can be converted to a floating point type, only integer values can be converted to an integer type. Why not permit a floating point value to be converted to an integer type, particularly since "all numbers are comparable" in ANSI SQL? The ANSI comparability statement notwithstanding, ANSI SQL does not permit approximate numeric values (the analog of Ada/SQL floating point) to be assigned to exact numeric (a superset of Ada/SQL integers) database columns or program variables. To enforce this restriction in Ada/SQL, CONVERT_TO is not allowed from a floating point value to an integer type. The net result is that any computations involving both integer and floating point values must be done in a floating point type. This would most likely be the desired mode anyway, due to the possible loss of precision when using integer arithmetic.

    Note that the CONVERT_TO prohibition on converting floating point to integer is in contrast to Ada type conversion (including as discussed with <value specification>), in which floating point values may be converted to an integer type.

11) Ada/SQL SR9 pertains to tables named with the same <table identifier> but with different <authorization identifier>s. Consider column c declared in table t1 with <authorization identifier> a1, and column c declared in table t1 with <authorization identifier> a2. An implementation which generates all effective Ada declarations must generate, for example, the following:

```
type VALUE_EXPRESSION_UNTYPED_a1_t is
  record
    . . .
    c : VALUE_EXPRESSION;
    . . .
  end record;

function t return VALUE_EXPRESSION_UNTYPED_a1_t;

type VALUE_EXPRESSION_UNTYPED_a2_t is
  record
```

```
    . . .
    c : VALUE_EXPRESSION;
    . . .
  end record;

function t return VALUE_EXPRESSION_UNTYPED_a2_t;
```

In a context where a value of effective type VALUE_EXPRESSION is required, as in the <select statement> fragment violating SR9 shown below, the required effective function t to use for the <column specification> t.c is not uniquely determined:

```
SELEC ( t.c & . . .
```

To comply with SR9, an <authorization identifier> must be used within the <table name> of the <column specification>, as in:

```
SELEC ( a1.t.c & . . .
```

12) Consider column c declared in table t1 of integer data type ct1, and column c declared in table t2 of integer data type ct2, which is different from ct1. An implementation which generates all effective Ada declarations must generate, for example, the following two functions:

```
function "<" ( LEFT : VALUE_EXPRESSION_ct1 ; RIGHT : ct1 )
 return SEARCH_CONDITION;

function "<" ( LEFT : VALUE_EXPRESSION_ct2 ; RIGHT : ct2 )
 return SEARCH_CONDITION;

function c return VALUE_EXPRESSION_ct1;

function c return VALUE_EXPRESSION_ct2;
```

Now, since 0 is a valid literal of both types ct1 and ct2, the required effective functions in the following <search condition>, which violates SR10, are not uniquely determined:

```
. . . c < 0 . . .
```

To comply with SR10, a <qualifier> must be used in the <column specification>, as in:

```
. . . t1.c < 0 . . .
```

13) Unless the database supports subtype checking, it is possible to use CONVERT_TO on a value not belonging to the subtype denoted by the <type identifier>; requiring checking for this condition could have an unacceptable performance impact on an Ada/SQL system. For this reason, GR3 states that programs performing bogus type conversions are erroneous. An implementation that can support database subtype checking may raise the DATA_EXCEPTION exception upon detecting a subtype constraint violation.

Note that Ada/SQL character string conversion includes a type conversion for each character in the string. This is in contrast to Ada type conversion for strings (including that discussed with <value specification>), which requires that two string types have the same component type in order to be

mutually convertible. The extended Ada/SQL convertibility is provided to match the functionality of ANSI SQL, in which all character strings are comparable.

5.8 <set function specification>

Function

Specify a value derived by the application of a function to an argument.

Format

<set function specification> ::=
COUNT ('*') | COUNT_ALL ('*')
| <distinct set function> | <all set function>

<distinct set function> ::=
{ AVG_DISTINCT | MAX_DISTINCT | MIN_DISTINCT | SUM_DISTINCT
| COUNT_DISTINCT } (<column specification>)

<all set function> ::=
{ AVG | MAX | MIN | SUM | AVG_ALL | MAX_ALL | MIN_ALL | SUM_ALL }
(<value expression>)

Effective Ada Declarations

```
-- see 8.10 for declaration of type STAR_TYPE

-- VALUE_EXPRESSION_DATABASE_INT is defined for predefined type DATABASE.INT
-- in accordance with 5.9 - For a program data type ct:
--    type VALUE_EXPRESSION_ct is private;

function COUNT ( STAR : STAR_TYPE ) return VALUE_EXPRESSION;

function COUNT ( STAR : STAR_TYPE ) return VALUE_EXPRESSION_INTEGER;

function COUNT ( STAR : STAR_TYPE ) return VALUE_EXPRESSION_DATABASE_INT;

function COUNT_ALL ( STAR : STAR_TYPE ) return VALUE_EXPRESSION
 renames COUNT;

function COUNT_ALL ( STAR : STAR_TYPE ) return VALUE_EXPRESSION_INTEGER
 renames COUNT;

function COUNT_ALL ( STAR : STAR_TYPE )
 return VALUE_EXPRESSION_DATABASE_INT renames COUNT;

function AVG_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION;

function AVG_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function AVG_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
```

```
 return VALUE_EXPRESSION;

function AVG_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_STRING )
 return VALUE_EXPRESSION;

function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ENUMERATION )
 return VALUE_EXPRESSION;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_STRING )
 return VALUE_EXPRESSION;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ENUMERATION )
 return VALUE_EXPRESSION;

function SUM_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION;

function SUM_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function SUM_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION;

function SUM_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;
```

```
function COUNT_DISTINCT ( COLUMN : COLUMN_SPECIFICATION )
 return VALUE_EXPRESSION;

function COUNT_DISTINCT ( COLUMN : COLUMN_SPECIFICATION )
 return VALUE_EXPRESSION_INTEGER;

function COUNT_DISTINCT ( COLUMN : COLUMN_SPECIFICATION )
 return VALUE_EXPRESSION_DATABASE_INT;

function AVG ( VALUE : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;

function AVG ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function AVG ( VALUE : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function AVG ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MAX ( VALUE : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;

function MAX ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function MAX ( VALUE : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function MAX ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MAX ( VALUE : VALUE_EXPRESSION_STRING ) return VALUE_EXPRESSION;

function MAX ( VALUE : VALUE_EXPRESSION_ENUMERATION )
 return VALUE_EXPRESSION;

function MIN ( VALUE : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;

function MIN ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function MIN ( VALUE : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function MIN ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function MIN ( VALUE : VALUE_EXPRESSION_STRING ) return VALUE_EXPRESSION;

function MIN ( VALUE : VALUE_EXPRESSION_ENUMERATION )
 return VALUE_EXPRESSION;

function SUM ( VALUE : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;

function SUM ( VALUE : VALUE_EXPRESSION_INTEGER )
```

```
 return VALUE_EXPRESSION_INTEGER;

function SUM ( VALUE : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function SUM ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function AVG_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION renames AVG;

function AVG_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER renames AVG;

function AVG_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION renames AVG;

function AVG_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING renames AVG;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION renames MAX;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER renames MAX;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION renames MAX;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING renames MAX;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_STRING )
 return VALUE_EXPRESSION renames MAX;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_ENUMERATION )
 return VALUE_EXPRESSION renames MAX;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION renames MIN;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER renames MIN;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION renames MIN;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING renames MIN;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_STRING )
 return VALUE_EXPRESSION renames MIN;
```

```
function MIN_ALL ( VALUE : VALUE_EXPRESSION_ENUMERATION )
 return VALUE_EXPRESSION renames MIN;

function SUM_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION renames SUM;

function SUM_ALL ( VALUE : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER renames SUM;

function SUM_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION renames SUM;

function SUM_ALL ( VALUE : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING renames SUM;
```

For an integer or floating point program data type ct:

```
function AVG_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ct )
 return VALUE_EXPRESSION_ct;

function SUM_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ct )
 return VALUE_EXPRESSION_ct;

function AVG ( VALUE : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function SUM ( VALUE : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function AVG_ALL ( VALUE : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct renames AVG;

function SUM_ALL ( VALUE : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct renames SUM;
```

For an integer, floating point, character string, or enumeration program data type ct:

```
function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ct )
 return VALUE_EXPRESSION_ct;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ct )
 return VALUE_EXPRESSION_ct;

function MAX ( VALUE : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function MIN ( VALUE : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct renames MAX;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct renames MIN;
```

For an enumeration data type ct that is not derived from another enumeration type (an ultimate parent type):

```
function MAX_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct;

function MIN_DISTINCT ( COLUMN : COLUMN_SPECIFICATION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct;

function MAX ( VALUE : VALUE_EXPRESSION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct;

function MIN ( VALUE : VALUE_EXPRESSION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct;

function MAX_ALL ( VALUE : VALUE_EXPRESSION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct renames MAX;

function MIN_ALL ( VALUE : VALUE_EXPRESSION_ENUMERATION_ct )
 return VALUE_EXPRESSION_ENUMERATION_ct renames MIN;
```

Example

```
NUMBER  : DATABASE.INT;
AVERAGE : EMPLOYEE_SALARY;
 . . .
SELEC ( COUNT ( '*' ), -- variation: COUNT_ALL
FROM => EMPLOYEE );
INTO ( NUMBER );
 . . .
SELEC ( COUNT_DISTINCT ( MANAGER ) ,
FROM => EMPLOYEE );
INTO ( NUMBER );
 . . .
SELEC ( AVG_DISTINCT ( SALARY ), -- variations: MAX_DISTINCT, MIN_DISTINCT,
FROM => EMPLOYEE );                             SUM_DISTINCT
INTO ( AVERAGE );
 . . .
SELEC ( AVG ( SALARY ), -- variations: MAX, MIN, SUM, AVG_ALL, MAX_ALL,
FROM => EMPLOYEE );                    MIN_ALL, SUM_ALL
INTO ( AVERAGE );
```

Syntax Rules

1) *The argument of* COUNT ('*') *or* COUNT_ALL ('*'), and the argument source of a <distinct set function> and <all set function> is a table or a group of a grouped table as specified in 5.19, "<table expression>", 5.24, "<subquery>", and 5.25, "<query specification>".

2) Let R denote the argument or argument source of a <set function specification>.

3) The <column specification> of a <distinct set function> and each <column specification> in the <value expression> of an <all set function> shall unambiguously reference a column of R and shall not reference a column derived from a <set function specification>.

4) The <value expression> of an <all set function> shall include a <column specification> that references a column of R and shall not include a <set function specification>. If the <column specification> is an outer reference, then the <value expression> shall not include any operators.

NOTE: "Outer reference" is defined in 5.7, "<column specification>".

5) If a <set function specification> contains a <column specification> that is an outer reference, then the <set function specification> shall be contained in a <subquery> of a <having clause>.

NOTE: "Outer reference" is defined in 5.7, "<column specification>".

6) Let T be the data type of the values that result from evaluation of the <column specification> or <value expression>.

7) If COUNT, COUNT_ALL, or COUNT_DISTINCT is specified, then the data type of the result of a <set function specification> is DATABASE.INT.

8) If MAX, MIN, MAX_ALL, or MIN_ALL is specified, then the data type of the result is T.

9) If SUM, AVG, SUM_ALL, or AVG_ALL is specified, then:

   a) T shall not be character string or enumeration.

   b) The data type of the result is T.

General Rules

1) The argument of a <distinct set function> is a set of values. The set is derived by the elimination of any null values and any redundant duplicate values from the column of R referenced by the <column specification>.

2) The argument of an <all set function> is a multi-set of values. The multi-set is derived by the elimination of any null values from the result of the application of the <value expression> to each row of R. The use or non-use of the _ALL suffix does not affect the meaning of an <all set function>.

3) Let S denote the argument of a <distinct set function> or an <all set function>.

4) Case:

   a) If the <distinct set function> COUNT_DISTINCT is specified, then the result is the cardinality of S.

b) If COUNT ('*') or COUNT_ALL ('*') is specified, then the result is the cardinality of R.

c) If AVG, MAX, MIN, or SUM (with or without _DISTINCT or _ALL suffix) is specified and S is empty, then the result is the null value.

d) If MAX or MIN (with or without _DISTINCT or _ALL suffix) is specified, then the result is respectively the maximum or minimum value in S. These results are determined using the comparison rules specified in 5.11, "<comparison predicate>".

e) If SUM (with or without _DISTINCT or _ALL suffix) is specified, then the result is the sum of the values in S. The sum shall be within the range of the base type of the result; otherwise, the program causing the <set function specification> to be evaluated is erroneous.

   i) An integer result shall be exact.

   ii) A floating point result shall be correct to the accuracy of its data type.

f) If AVG (with or without _DISTINCT or _ALL suffix) is specified, then the result is the average of the values in S. The sum of the values in S shall be within the range of the base type of the result; otherwise, the program causing the <set function specification> to be evaluated is erroneous.

   i) An integer result is carried forward to an implementor-defined number of decimal places, including at least all digits to the left of the decimal point. Such results may be used as "integer" operands to other operators, and the number of decimal places carried forward may affect the ultimate result of chains of operations. Any program whose effect depends on the number of decimal places carried forward is erroneous. When assigned to an integer <program variable> or database column, such an "integer" result is rounded to the nearest integer. A result that is halfway between two integers may be rounded either up or down. Any program whose effect depends on the direction of rounding of results halfway between two integers is erroneous.

   ii) A floating point result shall be correct to the accuracy of its data type.

Notes

1) The functions effectively declared for <set function specification> have four classes of return type, based on the context in which the <set function specification> appears. The class names and the contexts are the same as for <value expression>, as described in Note 1 of section 5.9, since <value expression> is the only production symbol containing <set function specification>. Lists of contexts in which each class of return type is effectively used for a <set function specification>, as immediately contained in a <value expression>, can therefore be found in that Note; only brief descriptions of those contexts are given here. The four classes of return type, and the contexts in which they are used, are:

   VALUE_EXPRESSION - used in contexts where the data type of the <set function specification> is not important for the effective Ada declarations.

VALUE_EXPRESSION_ct (typed according to program type) - used in contexts where the result of the <set function specification> will be used in another operation for which the effective Ada declarations are defined with strongly-typed operands.

VALUE_EXPRESSION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used in contexts where the result of the <set function specification> is the operand of a CONVERT_TO.

VALUE_EXPRESSION_y, where y is INTEGER or FLOATING - used in contexts where the result of the <set function specification> must be of a numeric (integer or floating point) data type because it is used as the operand of a monadic + or - operator, whose context is such that the function effectively declared for it returns a result of type VALUE_EXPRESSION or VALUE_EXPRESSION_y. Note that the VALUE_EXPRESSION_INTEGER and VALUE_EXPRESSION_FLOATING types are used for both this class and the previous class. Also note that this class, as defined for <value expression> in Note 1 of 5.9, also includes types VALUE_EXPRESSION_STRING and VALUE_EXPRESSION_ENUMERATION. These types are applicable only to a <value expression> contained in an <all set function>. Although a general <value expression> may be contained in an <all set function>, a <set function specification> cannot be, since SQL prohibits nesting of <set function specification>s. Thus, these latter types are not applicable to <set function specification>s.

Although the return types contained in class VALUE_EXPRESSION_y for <set function specification>s are a subset of those contained in class VALUE_EXPRESSION_x, the two classes are discussed separately below, because parameters to the effective Ada functions declared for <set function specification>s can be of all types listed for VALUE_EXPRESSION_y in Note 1 of 5.9. Thus, the class of the type, as determined by the context in which the <set function specification> appears, is important to the discussion, even if the same actual effective subprogram is used for two different classes. For example, a single effective <set function specification> subprogram may be applicable to type VALUE_EXPRESSION_INTEGER, whether that type is considered to be of class VALUE_EXPRESSION_x or VALUE_EXPRESSION_y.

2) The overloaded functions effectively declared for <set function specification>s have eight different types of operands. The effective type of the function parameter is based on the text of the corresponding <set function specification> and its context, as follows (the context is indicated in terms of the return class of the effective function, as described in Note 1):

STAR_TYPE - used for COUNT ('*') and COUNT_ALL ('*'); return type of any of the four classes; the type of the result of these counts is DATABASE.INT

COLUMN_SPECIFICATION_x - used for <distinct set function>s other than COUNT_DISTINCT where the result type will be of class VALUE_EXPRESSION_x (note that the effective parameter type reflects the fact that the argument to a <distinct set function> is restricted to be only a <column specification>)

COLUMN_SPECIFICATION_y - used for <distinct set function>s other than COUNT_DISTINCT where the result type will be of class VALUE_EXPRESSION or VALUE_EXPRESSION_y

COLUMN_SPECIFICATION_ct - used for <distinct set function>s other than COUNT_DISTINCT where the result type will be of class VALUE_EXPRESSION_ct

COLUMN_SPECIFICATION - used for COUNT_DISTINCT; return type of any of the four classes - the parameter to COUNT_DISTINCT does not require strong typing because the type of the result is always DATABASE.INT

VALUE_EXPRESSION_x - used for <all set function>s where the result type will be of class VALUE_EXPRESSION_x

VALUE_EXPRESSION_y - used for <all set function>s where the result type will be of class VALUE_EXPRESSION or VALUE_EXPRESSION_y

VALUE_EXPRESSION_ct - used for <all set function>s where the result type will be of class VALUE_EXPRESSION_ct

3) The Ada/SQL <set function specification> conforms to the ANSI SQL <set function specification>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	4
SR2-SR6	SR2-SR6	-
SR7	SR7	5
SR8	SR8	-
SR9	SR9	6
GR1-GR3	GR1-GR3	-
GR4	GR4	7

4) COUNT_ALL is provided as a synonym for COUNT because the COUNT name may conflict with the identical type names declared in TEXT_IO and DIRECT_IO.

5) To apply Ada/SQL's strong typing to the result of a COUNT, COUNT_ALL, or COUNT_DISTINCT operation, it is necessary to specify a data type for the result. This is always taken to be DATABASE.INT, which is guaranteed to have a range large enough to accommodate any possible database counts.

6) The ANSI prohibition on applying SUM or AVG to character strings is extended to enumeration data types in Ada/SQL. Ada/SQL SR9 includes specification of strong typing.

7) ANSI SQL allows the precision of SUM and AVG operations to be implementor-defined; Ada/SQL also allows this, but requires that the precision be at least that of the data type for which the operation is being performed.

ANSI SQL GR4 requires that the sums used in computing SUM and AVG be within the range of the data type of the result. Since Ada/SQL's concept of a data type includes a possible subtype constraint, we merely require that the sum be within the range of the base type, which does not include the subtype constraint. Implementations may raise the DATA_EXCEPTION exception if they

detect a sum out of range; we say that a program causing such an error is erroneous because there may be implementations that cannot readily detect the error.

ANSI SQL does not specify the precision of the result of an integer AVG. If such values are used as operands to other arithmetic operations, the final result may differ depending on the accuracy to which intermediate results are carried; see Note 17 in 5.9. Since ANSI SQL does not require any particular precision for the result of an AVG, Ada/SQL cannot either, although we do require that the result be correct to at least the nearest integer. We also note that a program whose effect depends on the precision of a particular implementation is erroneous.

8) Release 1 implementations do not support <distinct set function>s or the COUNT_ALL synonym for COUNT.

5.9 <value expression>

Function

Specify a value.

Format

<value expression> ::=
  [+ | -] <term>
 | <value expression> + <term>
 | <value expression> - <term>

<term> ::=
  <factor>
 | <term> * <factor>
 | <term> / <factor>

<factor> ::= <primary>

<primary> ::=
  <value specification>
 | <column specification>
 | <set function specification>
 | [CONVERT_TO . <library package name> . <type identifier>]
   (<value expression>)

Effective Ada Declarations

```
type VALUE_EXPRESSION is private;

type VALUE_EXPRESSION_INTEGER is private;

type VALUE_EXPRESSION_FLOATING is private;

type VALUE_EXPRESSION_STRING is private;

type VALUE_EXPRESSION_ENUMERATION is private;

function "+" ( LEFT : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;

function "+" ( LEFT : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function "+" ( LEFT : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function "+" ( LEFT : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;

function "-" ( LEFT : VALUE_EXPRESSION_INTEGER ) return VALUE_EXPRESSION;
```

```
function "-" ( LEFT : VALUE_EXPRESSION_INTEGER )
 return VALUE_EXPRESSION_INTEGER;

function "-" ( LEFT : VALUE_EXPRESSION_FLOATING ) return VALUE_EXPRESSION;

function "-" ( LEFT : VALUE_EXPRESSION_FLOATING )
 return VALUE_EXPRESSION_FLOATING;
```

For a program data type ct:

```
type VALUE_EXPRESSION_ct is private;
```

For an enumeration data type ct that is not derived from another enumeration type (an ultimate parent type):

```
type VALUE_EXPRESSION_ENUMERATION_ct is private;
```

For an integer or floating point program data type ct:

```
function "+" ( LEFT : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function "-" ( LEFT : VALUE_EXPRESSION_ct ) return VALUE_EXPRESSION_ct;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_ct;

function "+" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION;

function "+" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_ct;

function "-" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;
```

```
function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION;

function "-" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_ct;

function "*" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION;

function "*" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_ct;

function "/" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_ct;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION;

function "/" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION;
```

For an integer program data type ct:

```
function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_INTEGER;

function "+" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;
```

```
function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_INTEGER;

function "-" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_INTEGER;

function "*" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_INTEGER;

function "/" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_INTEGER;
```

For a floating point program data type ct:

```
function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "+" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_FLOATING;

function "+" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "-" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_FLOATING;

function "-" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "*" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_FLOATING;
```

```
function "*" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;

function "/" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return VALUE_EXPRESSION_FLOATING;

function "/" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return VALUE_EXPRESSION_FLOATING;
```

For an integer program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
     return VALUE_EXPRESSION_dt;

    function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
     return VALUE_EXPRESSION;

    function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
     return VALUE_EXPRESSION_INTEGER;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

For a floating point program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function ct ( LEFT : VALUE_EXPRESSION_FLOATING )
     return VALUE_EXPRESSION_dt;

    function ct ( LEFT : VALUE_EXPRESSION_FLOATING )
     return VALUE_EXPRESSION;

    function ct ( LEFT : VALUE_EXPRESSION_FLOATING )
     return VALUE_EXPRESSION_FLOATING;

    function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
     return VALUE_EXPRESSION_dt;

    function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
```

```
      return VALUE_EXPRESSION;

      function ct ( LEFT : VALUE_EXPRESSION_INTEGER )
       return VALUE_EXPRESSION_FLOATING;
       . . .
    end p;
     . . .
  end CONVERT_TO;
```

For a character string program subtype ct defined in library package p, of data type dt (ct may be the same as dt):

```
  package CONVERT_TO is
     . . .
    package p is
       . . .
      function ct ( LEFT : VALUE_EXPRESSION_STRING )
       return VALUE_EXPRESSION_dt;

      function ct ( LEFT : VALUE_EXPRESSION_STRING )
       return VALUE_EXPRESSION;

      function ct ( LEFT : VALUE_EXPRESSION_STRING )
       return VALUE_EXPRESSION_STRING;
       . . .
    end p;
     . . .
  end CONVERT_TO;
```

For an enumeration program subtype ct defined in library package p, of data type dt with ultimate parent type pt (ct may be the same as dt, and dt may be the same as pt):

```
  package CONVERT_TO is
     . . .
    package p is
       . . .
      function ct ( LEFT : VALUE_EXPRESSION_ENUMERATION_pt )
       return VALUE_EXPRESSION_dt;

      function ct ( LEFT : VALUE_EXPRESSION_ENUMERATION_pt )
       return VALUE_EXPRESSION;

      function ct ( LEFT : VALUE_EXPRESSION_ENUMERATION_pt )
       return VALUE_EXPRESSION_ENUMERATION;

      function ct ( LEFT : VALUE_EXPRESSION_ENUMERATION_pt )
       return VALUE_EXPRESSION_ENUMERATION_pt;
       . . .
    end p;
     . . .
  end CONVERT_TO;
```

Example

```
MINIMUM_WAGE    : EXAMPLE_TYPES.ADA_SQL.HOURLY_WAGE;
ESTIMATED_TIPS  : EMPLOYEE_SALARY;
NUMBER          : DATABASE.INT;
 . . .
SELEC  ( COUNT ( '*' ),
FROM   => EMPLOYEE,
WHERE  => CONVERT_TO.EXAMPLE_TYPES.HOURLY_WAGE
          ( ( SALARY + ESTIMATED_TIPS ) / 2080.0 ) < + MINIMUM_WAGE );
INTO ( NUMBER );
                  -- variations: - vs. + binary (dyadic) adding operator
                  --             * vs. / multiplying operator
                  --             - vs. + unary (monadic) adding operator
```

Syntax Rules

1) A <value expression> that includes a <distinct set function> shall not include any dyadic operators.

2) If the <primary> is of a character string or an enumeration data type, then the <value expression> shall not include any arithmetic operators. The data type of the result is the same as that of the <primary>.

3) The data type of the result of a monadic arithmetic operator is the same as the data type of the (integer or floating point) <term> to which it is applied.

4) Case:

   a) If both operands of a dyadic arithmetic operator are of a universal data type, then:

      Case:

      i) If both operands are of the same universal data type (universal integer or universal floating point), then the data type of the result is the same as that of the operands.

      ii) If one operand is of the universal integer data type and the other operand is of the universal floating point data type, then the result is of the universal floating point data type, and one of the following shall be true:

         1) The operator shall be multiplication, or

         2) The operator shall be division, and the right operand shall be the one of the universal integer data type.

   b) If either operand of a dyadic arithmetic operator is not of a universal data type, then both operands shall be of the same data type. The data type of the result is the same as that of the

operands.

5) The data type of the result of a CONVERT_TO is that denoted by the <type identifier>, and shall be an integer, floating point, character string, or enumeration data type.

Case:

a) If the <library package name> is STANDARD, then the <type identifier> shall be declared within the STANDARD Ada/SQL predefined environment.

b) If the library package denoted by the <library package name> is part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within that library package.

c) If the library package denoted by the <library package name> is not part of the Ada/SQL predefined environment, then the <type identifier> shall be declared within the ADA_SQL nested package of that library package.

6) The <value expression> operand of a CONVERT_TO shall contain at least one of the following: a <column specification>; a <set function specification>; the <key word> USER; a <program object name>, other than a <named number>, not contained in an <indicator value>; an <Ada type qualification>; or an <enumeration literal> which is a literal of exactly one enumeration data type declared in a <schema package> or the predefined Ada/SQL environment.

7) The <value expression> operand of a CONVERT_TO shall contain at least one of the following: a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER.

8) Case:

a) If the <type identifier> of a CONVERT_TO denotes an integer data type, then the <value expression> operand of the CONVERT_TO shall be of an integer data type.

b) If the <type identifier> of a CONVERT_TO denotes a floating point data type, then the <value expression> operand of the CONVERT_TO shall be of an integer or a floating point data type.

c) If the <type identifier> of a CONVERT_TO denotes a character string data type, then the <value expression> operand of the CONVERT_TO shall be of a character string data type.

d) If the <type identifier> of a CONVERT_TO denotes an enumeration data type, then the <value expression> operand of the CONVERT_TO shall be of an enumeration data type such that both enumeration data types have the same ultimate parent type.

General Rules

1) If the value of any <primary> is the null value, then the result of the <value expression> is the null value.

2) The result of a CONVERT_TO with a <value expression> equal to the null value is the null value. If the subtype denoted by the <type identifier> does not permit null values, then the program executing the CONVERT_TO is erroneous.

3) The result of a nonnull CONVERT_TO is the value of its <value expression> operand, typed according to the <type identifier>. If the result of the <value expression> does not belong to the subtype denoted by the <type identifier>, then the program executing the CONVERT_TO is erroneous.

   Case:

   a) Conversion of an integer value to an integer value shall be exact.

   b) Conversion of an integer or a floating point value to a floating point value shall retain at least the accuracy of the subtype denoted by the <type identifier>.

   c) Conversion of a character string value to a character string value shall be as follows:

      i) The result character string shall have as many characters as the <value expression> character string.

         Case:

         1) If the subtype denoted by the <type identifier> is an unconstrained character string, then the maximum number of characters in a string of that subtype shall not be less than the number of characters in the character string <value expression>, otherwise the program executing the CONVERT_TO is erroneous.

         2) If the subtype denoted by the <type identifier> is a constrained character string, then the number of characters in that string subtype shall be the same as the number of characters in the character string <value expression>, otherwise the program executing the CONVERT_TO is erroneous.

      ii) Successive characters in the result are set to successive characters in the <value expression>, converted to the component data type of the character string data type denoted by the <type identifier>. If any character in the <value expression> does not belong to the component subtype, then the program executing the CONVERT_TO is erroneous.

   d) Conversion of an enumeration value to another enumeration value shall be according to matching enumeration literals.

4) If operators are not specified, then the result of the <value expression> is the value of the specified <primary>.

5) When a <value expression> is applied to a row of a table, each reference to a column of that table is a reference to the value of that column in that row.

6) The monadic arithmetic operators + and - specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand. Except where the result is of a universal data type, the result of a monadic operator shall belong to the base type of its operand; otherwise the program causing the <value expression> to be evaluated is erroneous.

7) The dyadic arithmetic operators +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. A divisor shall not be 0; otherwise, the DATA_EXCEPTION exception is raised. Except where the result is of a universal data type, the result of a dyadic arithmetic operator shall belong to the base type of its operands; otherwise, the program causing the <value expression> to be evaluated is erroneous.

8) All arithmetic operators shall yield mathematically correct results.

   a) The result of integer operations other than division shall be exact.

   b) The result of integer division is carried forward to an implementor defined number of decimal places, including at least all digits to the left of the decimal point. Such results may be used as "integer" operands to other operators, and the number of decimal places carried forward may affect the ultimate result of chains of operations. Any program whose effect depends on the number of decimal places carried forward is erroneous. When assigned to an integer <program variable> or database column, such an "integer" result is rounded to the nearest integer. A result that is halfway between two integers may be rounded either up or down. Any program whose effect depends on the direction of rounding of results halfway between two integers is erroneous.

   c) The result of floating point operations shall be correct to the accuracy of the data type of the result.

9) Expressions within parentheses are evaluated first and when the order of evaluation is not specified by parentheses, multiplication and division are applied before monadic operators, monadic operators are applied before addition and subtraction, and operators at the same precedence level are applied from left to right.

Notes

1) The functions effectively declared for <value expression> have four classes of return type, based on the context in which the <value expression> appears, as follows:

   VALUE_EXPRESSION - used in contexts where the data type of the <value expression> is not important for the effective Ada declarations. Relevant contexts immediately containing <value expression>:

   <subquery>, used in the following context: <exists predicate>

<select list>

VALUE_EXPRESSION_ct (typed according to program type) - used in contexts where the result of the <value expression> will be used in another operation for which the effective Ada declarations are defined with strongly-typed operands. Relevant contexts immediately containing <value expression>:

<value expression>, when used as the left operand of a dyadic + or -

<primary>, when used as a <term> or a <factor> that is an operand of a dyadic +, -, *, or /

<comparison predicate>

<between predicate>

<in predicate>

<quantified predicate>

<subquery> used in the following contexts: <comparison predicate>, <in predicate>, <quantified predicate>

<set clause: positioned>

<set clause: searched>

VALUE_EXPRESSION_x, where x is INTEGER, FLOATING, STRING, or ENUMERATION_ct, where ct is the name of an enumeration data type not derived from any other enumeration type (an ultimate parent type) - used in contexts where the <value expression> is the operand of a CONVERT_TO. See Note 2.

VALUE_EXPRESSION_y, where y is INTEGER, FLOATING, STRING, or ENUMERATION - used in contexts where the result of the <value expression> will be used in an operation with only one operand, where the type class (integer, floating point, character string, or enumeration) of the operand is important, and where the context of that operation is such that the function effectively declared for it returns a result of type VALUE_EXPRESSION or VALUE_EXPRESSION_y. Note that three of the same types (VALUE_EXPRESSION_INTEGER, VALUE_EXPRESSION_FLOATING, and VALUE_EXPRESSION_STRING) are used for both this class and the previous class. See also Note 3. Relevant contexts immediately containing <value expression> (these contexts may also require functions returning strongly-typed VALUE_EXPRESSION_ct, if they are themselves contained in a context where the effective Ada declarations are defined with strongly-typed operands - see Note 4):

<all set function>

<primary>, when used as a <term> that is an operand of a monadic + or - operator

Effective Ada/SQL subprograms with parameters corresponding to <value expression>s are also declared with those parameters of the appropriate program data types (denoted as "ct" in this document). A <value expression> not containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER will be of a program type, rather than of one of the Ada/SQL return types described above. Any operators applied to

program values are, of course, standard Ada operators, rather than the effective Ada/SQL operators discussed here.

2) There are several CONVERT_TO functions effectively declared for each subtype, ct, to which a value may be converted. The effective type of the parameter to each of these functions denotes the classes of data types that may be converted to the target data type. Thus, only integer values may be converted to an integer data type, either integer or floating point values may be converted to a floating point data type, and only string values may be converted to a string data type. Enumeration data types are themselves divided into classes, since (by Ada rules) a conversion is allowed from an operand type to a target type if one of the two types is derived from the other, directly or indirectly, or if there exists a third type from which both types are derived, directly or indirectly. This enables enumeration types to be partitioned into classes, with types in the same class being mutually convertible while those in different classes are not. Within each class there exists a data type that is not a derived type; all other types in the class are derived, directly or indirectly, from it. We use this so-called ultimate parent type to designate the class.

   We have elected to type the parameter of effective CONVERT_TO functions according to the appropriate data type class. It would also be feasible to strongly type the parameter according to the actual data type of the operand being converted. An implementation which generates all possible effective Ada/SQL declarations based on the type declarations contained in a <schema> would then have to generate order(n**2) functions, however, where n is the number of different data types declared. By using type classes, the number of functions that must be generated is order(n).

3) Those functions that are not strongly typed that are effectively declared for the monadic operators also have a parameter typed according to a class of data types. Since no conversion is involved here, it is sufficient to distinguish integer, floating point, string, and enumeration, without dividing the enumeration types into classes. Actually, it is really only necessary to distinguish numeric and non-numeric, but we elected to reuse some of the effective types invented for the CONVERT_TO functions. This yields a smaller number of possible effective functions than would defining new effective types such as VALUE_EXPRESSION_NUMERIC and VALUE_EXPRESSION_NON_-NUMERIC. It would also be possible to always strongly type the parameter to monadic operators, even in contexts where the result should not be strongly typed. Our decision to only strongly type the parameter if the result is to be strongly typed again reduces the total number of possible effective functions required.

4) Effective monadic operators returning a strongly typed result have a strongly typed parameter, (type denoted by VALUE_EXPRESSION_ct in Note 1). Effective monadic operators are not defined with parameter of program data type ct, because the standard Ada operators apply in that case. Effective monadic operators returning a result indicative of type class or a VALUE_EXPRESSION result have a parameter typed according to type class (type denoted by VALUE_EXPRESSION_y in Note 1), rather than being strongly typed. Example of monadic - with strongly typed parameter (assume A, B, and C are database columns):

   . . . - A * B - C . . .

   Parameters to dyadic -, both strongly typed: - A * B and C

   Parameter to monadic -, strongly typed: A * B

   Parameters to dyadic *, both strongly typed: A and B

Example of monadic - with parameter and result typed according to class (assume A is a database column, ct is a numeric subtype defined in library package p):

```
. . . CONVERT_TO.p.ct ( - A ) . . .
```

Example of monadic - with parameter typed according to class and result of type VALUE_EXPRESSION (assume A is a database column):

```
SELEC ( - A & . . . (<select statement>)
```

5) The overloaded dyadic operators have two different types of operands. The effective type of each operand is based on the text of the corresponding <value expression>, <term>, or <factor> in the <value expression>, as follows:

VALUE_EXPRESSION_ct (typed according to program type) - used when the corresponding construct contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

ct (program type) - used when the corresponding construct does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

6) The Ada/SQL <value expression> conforms to the ANSI SQL <value expression>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	-
SR2	—	7
SR3	SR2	8
—	SR3	9
SR4-SR5	SR4	10
—	SR5	11
—	SR6	12
—	SR7	13
—	SR8	14
GR1	GR1	-
—	GR2-GR3	15
GR2-GR3	GR4-GR5	-
GR4-GR5	GR6-GR7	16
GR6	GR8	17
GR7	GR9	18

7) ANSI SQL SR2, prohibiting a plus or a minus sign from being adjacent to another plus or minus sign, without an intervening token, is not necessary in Ada/SQL. The Ada/SQL BNF does not permit such an occurrence in non-comment program text, so an explicit SR is not necessary. (The

reason that the adjacency is possible in ANSI SQL is that an ANSI SQL <numeric literal> can begin with a leading plus or minus sign. Ada numeric literals may not have a leading sign, so neither can Ada/SQL <numeric literal>s. An Ada/SQL comment delimiter ("--"), which starts with a minus sign, may follow a plus or minus sign in an Ada/SQL statement; the corresponding ANSI SQL statement does not include the comment or its delimiter.)

8) The ANSI prohibition on applying arithmetic operators to character strings is extended to enumeration types in Ada/SQL. Ada/SQL SR2 includes specification of strong typing.

9) There is no ANSI SR that defines the data type of the result of a monadic operator, but it is presumably the same as the data type of the operand. Ada/SQL explicitly specifies the data type, in accordance with strong typing.

10) Ada/SQL SR4 expresses one aspect of Ada/SQL's strong typing, and also embodies Ada rules for handling numbers of type universal_integer and universal_real. See also Note 17 for comments on the accuracy of integer division.

11) Based on the SDL syntax rules, the expanded name of a user-defined subtype denoted by a <type identifier> is <library package name>.ADA_SQL.<type identifier>. The designation of a CONVERT_TO for that subtype is, however, CONVERT_TO.<library package name>.<type identifier>, omitting the ADA_SQL. Omitting the ADA_SQL does not introduce any ambiguities, because the only <type identifier>s contained in the named library package that may be referenced in a CONVERT_TO are those defined in the ADA_SQL nested package.

12) Ada/SQL SR6 ensures that the data type of the operand to the CONVERT_TO is known from the source text, similar to the Ada type conversion requirement that "the type of the operand of a type conversion must be determinable independently of the context". There is an additional reason for this restriction, however. Effective functions required for the operands to CONVERT_TOs used within a program may be ambiguous when applied to an operand whose data type is not explicitly known. For example, consider a floating point data type, TAX_AMOUNT, declared in library package P, two other floating point data types, ANNUAL_SALARY and MONTHLY_SALARY, two program variables, ANNUAL_PAY and MONTHLY_PAY, of those two data types, respectively, and a program containing the following two <select statement>s (the right operand to the multiplication is presumably a lookup of tax rates in a database table):

```
SELEC ( CONVERT_TO.P.TAX_AMOUNT ( INDICATOR ( ANNUAL_PAY ) ) * . . .

SELEC ( CONVERT_TO.P.TAX_AMOUNT ( INDICATOR ( MONTHLY_PAY ) ) * . . .
```

An implementation which actually generates the effective Ada declarations must generate at least the following:

```
package CONVERT_TO is
  package P is
    function TAX_AMOUNT ( LEFT : VALUE_EXPRESSION_FLOATING )
     return VALUE_EXPRESSION_TAX_AMOUNT;
  end P;
end CONVERT_TO;
```

```
function INDICATOR
         ( VALUE     : ANNUAL_SALARY;
           INDICATOR : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_FLOATING;

function INDICATOR
         ( VALUE     : MONTHLY_SALARY;
           INDICATOR : INDICATOR_VARIABLE := NOT_NULL )
 return VALUE_EXPRESSION_FLOATING;
```

Now, suppose the same program also contains the following <select statement>, in violation of SR6:

```
SELEC ( CONVERT_TO.P.TAX_AMOUNT ( INDICATOR ( 25_000.00 ) ) * . . .
```

25_000.00 is a literal value of both ANNUAL_SALARY and MONTHLY_SALARY data types, so that the effective INDICATOR function is ambiguous. To comply with SR6, the type of the literal can be explicitly qualified, as in the following legal fragment:

```
SELEC ( CONVERT_TO.P.TAX_AMOUNT
        ( INDICATOR ( ANNUAL_SALARY' ( 25_000.00 ) ) ) & . . .
```

Release 1 implementations do not enforce SR6 if the <value expression> is of an integer, floating point, or character string data type. Instead, the type of the <value expression> is assumed to be STANDARD.INTEGER, STANDARD.FLOAT, or STANDARD.STRING, as appropriate.

13) A <value expression> not containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If CONVERT_TO were allowed to data type t1, defined in library package p, from such a <value expression> of data type t2, then the following CONVERT_TO function (as well as others) must be effectively declared:

```
package CONVERT_TO is
   . . .
  package p is
     . . .
    function t1 ( LEFT : t2 ) return VALUE_EXPRESSION_t1;
     . . .
  end p;
   . . .
end CONVERT_TO;
```

An implementation which generates all possible effective Ada/SQL declarations, based on the type declarations contained in a <schema>, would have to generate order(n**2) such functions, where n is the number of different data types declared. By prohibiting the <value expression> operand of a CONVERT_TO from being of a program type, SR7 ensures that the number of functions effectively declared is linear in the number of user data types defined, rather than depending on the square of the number of data types. Note that the effective Ada/SQL declarations are such that a <value expression> containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER, when used as the operand to a CONVERT_TO, effectively returns an object of one of the VALUE_EXPRESSION_x types described in Note 1,

rather than a type unique to its underlying conceptual type, thereby avoiding the n**2 problem.

Release 1 implementations do not support the <key word> USER.

The restriction of Ada/SQL SR7 should actually have virtually no impact on Ada/SQL programmers, because <Ada type conversion>s (described with <value specification>) can be used on the prohibited <value expression>s. CONVERT_TO does provide more flexibility on character string conversions than does Ada type conversion (see Note 15). To use this flexibility on program objects, the program object can be expressed as an<indicator specification> to comply with Ada/SQL SR7. For example, the following violates SR7 (assume that SOCIAL_SECURITY_NUMBER is a program variable of a character string type, with components of a data type other than those of character string type IDENTIFICATION_NUMBER, defined in package P):

```
. . . CONVERT_TO.P.IDENTIFICATION_NUMBER
      ( SOCIAL_SECURITY_NUMBER ) . . .
```

However, the following has the exact same effect, and is permitted:

```
. . . CONVERT_TO.P.IDENTIFICATION_NUMBER
      ( INDICATOR ( SOCIAL_SECURITY_NUMBER ) ) . . .
```

14) CONVERT_TO is designed such that only types of the same class are mutually convertible: numerics to numerics, character strings to character strings, and enumerations to enumerations with the same ultimate parent type. Numerics are not totally mutually convertible, however. Although both integer and floating point values can be converted to a floating point type, only integer values can be converted to an integer type. Why not permit a floating point value to be converted to an integer type, particularly since "all numbers are comparable" in ANSI SQL? The ANSI comparability statement notwithstanding, ANSI SQL does not permit approximate numeric values (the analog of Ada/SQL floating point) to be assigned to exact numeric (a superset of Ada/SQL integers) database columns or program variables. To enforce this restriction in Ada/SQL, CONVERT_TO is not allowed from a floating point value to an integer type. The net result is that any computations involving both integer and floating point values must be done in a floating point type. This would most likely be the desired mode anyway, due to the possible loss of precision when using integer arithmetic.

Note that the CONVERT_TO prohibition on converting floating point to integer is in contrast to Ada type conversion (including as discussed with <value specification>), in which floating point values may be converted to an integer type.

15) Unless the database supports subtype checking, it is possible to use CONVERT_TO on a value not belonging to the subtype denoted by the <type identifier>; requiring checking for this condition could have an unacceptable performance impact on an Ada/SQL system. For this reason, GR3 states that programs performing bogus type conversions are erroneous. An implementation that can support database subtype checking may raise the DATA_EXCEPTION exception upon detecting a subtype constraint violation.

Note that Ada/SQL character string conversion includes a type conversion for each character in the string. This is in contrast to Ada type conversion for strings (including that discussed with <value specification>), which requires that two string types have the same component type in order to be mutually convertible. The extended Ada/SQL convertibility is provided to match the functionality

of ANSI SQL, in which all character strings are comparable.

16) Ada/SQL GR6 and GR7 address the question of arithmetic results out of representable range. Since the capabilities of SQL implementations vary considerably, and requiring any specific checking on arithmetic results could have an unacceptable performance impact on an Ada/SQL system, it was felt best to simply declare programs erroneous if they performed a computation out of the range of the base type involved. An implementation that can support such error checking may raise the DATA_EXCEPTION exception upon detecting a result out of range of the required base type.

17) Although ANSI GR6 addresses the accuracy of SQL exact numeric operations, it does not address that of approximate numeric operations. Ada/SQL GR8 does address floating point operations, which are analogous to ANSI SQL approximate numeric operations. Ada/SQL integers are analogous to a subset of ANSI SQL's exact numerics. (Ada/SQL support may be extended at a later date to encompass all of ANSI SQL exact numerics.) ANSI SQL SR4 and GR6 together determine that integer addition, subtraction, and multiplication are exact. ANSI SQL SR4d (particularly) and GR6b, however, leave unspecified the accuracy of intermediate results of division. With integer arithmetic involving several operations, the final result may differ depending on the accuracy to which intermediate results are carried. Example:

2 / 3 + 2 / 3 = 1 + 1 = 2 if intermediate results are rounded to integer

= .7 + .7 = 1 if intermediate results are rounded to one decimal place

Since ANSI SQL does not require any particular implementation of intermediate results, Ada/SQL cannot either, although we do require that the result be correct to at least the nearest integer. We also note that a program whose effect depends on the precision of a particular implementation is erroneous.

Note that the requirement of ANSI SQL SR6a, that an exact arithmetic result be representable with the precision and scale (0 for Ada/SQL integers) of the result type, need not be stated for Ada/SQL, because it is implicit in the Ada/SQL requirement that the result belong to the same data type as the operands.

18) Ada/SQL GR9 describes the Ada order of expression evaluation, which is not the same as that prescribed in ANSI SQL GR7. The properties of the arithmetic operators are such, however, that results should be the same regardless of which order of evaluation is used.

5.10 <predicate>

Function

Specify a condition that can be evaluated to give a truth value of "true", "false", or "unknown".

Format

<predicate> ::=
    <comparison predicate>
  | <between predicate>
  | <in predicate>
  | <like predicate>
  | <null predicate>
  | <quantified predicate>
  | <exists predicate>

Effective Ada Declarations

None.

Example

See 5.11 - 5.17.

Syntax Rules

None.

General Rules

1) The result of a <predicate> is derived by applying it to a given row of a table.

Notes

1) The Ada/SQL <predicate> conforms to the ANSI SQL <predicate>.

2) Release 1 implementations do not support the <null predicate>, the <quantified predicate>, or the <exists predicate>.

5.11 <comparison predicate>

Function

Specify a comparison of two values.

Format

<comparison predicate> ::=
    <equality operator>
    (<value expression> , { <value expression> | <subquery> })
  | <value expression> <ordering operator>
    { <value expression> | <subquery> }

<equality operator> ::=
    EQ | NE

<ordering operator> ::=
    < | > | <= | >=

Effective Ada Declarations

For a program data type ct:

```
function EQ ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function EQ ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function EQ ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function EQ ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function EQ ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;

function NE ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function NE ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function NE ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function NE ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;
```

```
function NE ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;

function "<" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function "<" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function "<" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function "<" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function "<" ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;

function ">" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function ">" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function ">" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function ">" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function ">" ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;

function "<=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;

function ">=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;

function ">=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : ct )
 return SEARCH_CONDITION;

function ">=" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION_ct )
 return SEARCH_CONDITION;
```

```
function ">=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function ">=" ( LEFT : ct ; RIGHT : SUBQUERY_ct ) return SEARCH_CONDITION;
```

Example

```
package E is new EMPLOYEE_CORRELATION.NAME ( "E" );

CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => E.EMPLOYEE,
  WHERE => SALARY >                              -- variations: <, <=, >=
    SELEC  ( AVG ( SALARY ),
    FROM  => EMPLOYEE,
    WHERE => EQ ( MANAGER , E.MANAGER ) ) ) ); -- variation: NE
```

Syntax Rules

1) The data types of the first <value expression> and the <subquery> or second <value expression> shall be the same.

2) If the <comparison predicate> is of the form <value expression> <ordering operator> <value expression>, then at least one of the two <value expression>s shall contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER.

General Rules

1) Let x denote the result of the first <value expression> and let y denote the result of the <subquery> or the second <value expression>. The result of the <subquery> shall be at most one value; otherwise, the CARDINALITY_VIOLATION exception is raised.

2) If x or y is the null value or if the result of the <subquery> is empty, then the result of the <comparison predicate> is unknown.

3) If x and y are non-null values, then the <comparison predicate> is either true or false:

   "EQ (x , y)" is true if and only if x and y are equal.
   "NE (x , y)" is true if and only if x and y are not equal.
   "x < y" is true if and only if x is less than y.
   "x > y" is true if and only if x is greater than y.
   "x <= y" is true if and only if x is not greater than y.
   "x >= y" is true if and only if x is not less than y.

4) Integer and floating point numbers are compared with respect to their algebraic values.

5) Enumeration values are compared with respect to the ordering specified by the <enumeration type>.

6) The comparison of two character strings is determined by the comparison of <character>s with the same ordinal position. If the strings do not have the same length, then the comparison is made with a working copy of the shorter string that has been effectively extended on the right with <space>s so that it has the same length as the other string.

7) Two strings are equal if all <character>s with the same ordinal position are equal. If two strings are not equal, then their relation is determined by the comparison of the first pair of unequal <character>s from the left end of the strings. This comparison is made with respect to the ASCII collating sequence.

8) Although "EQ (x , y)" is unknown if both x and y are null values, in the contexts of GROUP_BY, ORDER_BY, and any <key word> suffixed with _DISTINCT, a null value is identical to or is a duplicate of another null value.

Notes

1) The overloaded comparison operators have three different types of operands. The effective type of each operand is based on the text of the <comparison predicate> as follows:

   VALUE_EXPRESSION_ct (typed according to program type) - corresponds to a <value expression> in the <comparison predicate>; used when the <value expression> contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   ct (program type) - corresponds to a <value expression> in the <comparison predicate>; used when the <value expression> does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   SUBQUERY_ct (typed according to program type) - corresponds to a <subquery> right operand in the <comparison predicate>

2) The Ada/SQL operators corresponding to ANSI SQL "=" and "<>" are written as prefix EQ and NE, due to Ada restrictions on overloading infix "=" and "/=".

3) The Ada/SQL <comparison predicate> conforms to the ANSI SQL <comparison predicate>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	4
—	SR2	5
GR1-GR4	GR1-GR4	-
—	GR5	6
GR5	GR6	7
GR6	GR7	8
GR7	GR8	-

4) Ada/SQL SR1 expresses one aspect of Ada/SQL's strong typing.

5) A <value expression> not containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If two such <value expression>s (necessarily of the same data type) were to be used in a <comparison predicate> separated by an <ordering operator>, then the effective Ada/SQL declaration for that <ordering operator> would redefine the corresponding predefined Ada operator for that program type. By prohibiting two <value expression>s separated by an <ordering operator> in a <comparison predicate> from both being of a program type, SR2 prevents this undesirable situation. Note that the effective Ada/SQL declarations are such that a <value expression> for program data type ct, containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER, when used in a <comparison predicate>, effectively returns an object of type VALUE_EXPRESSION_ct. Thus, the effective Ada/SQL declaration of the <ordering operator> with at least one parameter of type VALUE_EXPRESSION_ct does not redefine the corresponding predefined Ada operator with parameters of data type ct.

   The restriction of Ada/SQL SR2 should actually have virtually no impact on Ada/SQL programmers, because the result of a prohibited <comparison predicate> is not dependent on the database, and so could be computed outside of the Ada/SQL statement. If a programmer insists on using two <value expression>s not containing any of a <column specification>, a <set function specification>, or the <key word> USER in a <comparison predicate> separated by an <ordering operator>, then one of the <value expression>s can be expressed as an <indicator specification> to comply with Ada/SQL SR2. For example, the following violates SR2 (assume that PROPOSED_SALARY and CURRENT_SALARY are program variables of type EMPLOYEE_SALARY):

```
. . . PROPOSED_SALARY > CURRENT_SALARY . . .
```

   However, the following has the exact same effect, and is permitted:

```
. . . INDICATOR ( PROPOSED_SALARY ) > CURRENT_SALARY . . .
```

   Release 1 implementations do not support the <key word> USER.

6) Ada/SQL extends ANSI SQL to include support of enumeration types.

7) Note that ANSI SQL and Ada/SQL character string comparisons will yield different results from Ada string comparisons for two strings of unequal lengths that differ only in the number of trailing blanks. ANSI SQL and Ada/SQL consider such strings to be equal; Ada considers the shorter string to be less than the longer one.

8) The collating sequence for <character>s is implementor-defined in ANSI SQL; Ada/SQL requires that the collating sequence be ASCII, consistent with Ada.

5.12 <between predicate>

Function

Specify a range comparison.

Format

<between predicate> ::=
BETWEEN (<value expression> , <value expression> AND <value expression>)

Effective Ada Declarations

For a program data type ct:

```
type AND_ct is private;

function "AND"
         ( LEFT  : VALUE_EXPRESSION_ct;
           RIGHT : VALUE_EXPRESSION_ct ) return AND_ct;

function "AND" ( LEFT  : VALUE_EXPRESSION_ct ; RIGHT : ct ) return AND_ct;

function "AND" ( LEFT  : ct ; RIGHT : VALUE_EXPRESSION_ct ) return AND_ct;

function "AND" ( LEFT  : ct ; RIGHT : ct ) return AND_ct;

function BETWEEN ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : AND_ct )
 return SEARCH_CONDITION;

function BETWEEN ( LEFT : ct ; RIGHT : AND_ct ) return SEARCH_CONDITION;
```

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => BETWEEN ( SALARY , 20_000.00 AND 30_000.00 ) ) );
```

Syntax Rules

1) The data types of the three <value expression>s shall be the same.

2) If the <value expression>s are of a boolean data type, then at least one of the last two (separated by the AND) shall contain a <column specification> or an <indicator specification>.

General Rules

1) Let x, y, and z denote the result of the first, second, and third <value expression>, respectively.

2) "BETWEEN (x , y AND z)" has the same result as "x >= y AND x <= z".

Notes

1) The overloaded "AND" operators have two different types of operands. Likewise, there are two different types possible for the first parameter of the BETWEEN functions. The effective type of each operand is based on the text of the corresponding <value expression> in the <between predicate>, as follows:

   VALUE_EXPRESSION_ct (typed according to program type) - used when the <value expression> contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   ct (program type) - used when the <value expression> does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

2) The Ada/SQL <between predicate> conforms to the ANSI SQL <between predicate>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	3
—	SR2	4
GR1-GR2	GR1-GR2	-
GR3	—	5

3) Ada/SQL SR1 expresses one aspect of Ada/SQL's strong typing.

4) A boolean <value expression> containing neither a <column specification> nor an <indicator specification> is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If two such <value expression>s of a boolean program type were to be used as the last two <value expression>s in a <between predicate>, then the effective Ada/SQL declaration of the "AND" operator separating them would redefine the predefined boolean "and" operator for that program type. By prohibiting the last two <value expression>s in a <between predicate> from both being of a boolean program type, SR2 prevents this undesirable situation. Note that the effective Ada/SQL declarations are such that a <value expression> for program data type ct, containing a <column specification> or an <indicator specification>, when used in a <between predicate>, effectively returns an object of type VALUE_EXPRESSION_ct. Thus, the effective Ada/SQL declaration of an "AND" operator with at least one parameter of type VALUE_EXPRESSION_ct does not redefine the predefined "and" operator with parameters of data type ct.

   The restriction of Ada/SQL SR2 should actually have virtually no impact on Ada/SQL

programmers, because the limited range of boolean values makes their use in a <between predicate> extremely limited. If a programmer insists upon using boolean values in a <between predicate>, then one of the last two <value expression>s can be expressed as an <indicator specification> to comply with Ada/SQL SR2, if neither of these <value expression>s contain a <column specification>. For example, the following violates SR2 (assume that B1 and B2 are program variables of the same boolean data type, and that x is a <value expression> of that data type):

```
. . . BETWEEN ( x , B1 AND B2 ) . . .
```

However, the following has the exact same effect, and is permitted:

```
. . . BETWEEN ( x , INDICATOR ( B1 ) AND B2 ) . . .
```

5) Ada/SQL does not permit the NOT <key word> in a <between predicate>. Ada/SQL syntax does, however, permit the sense of a <between predicate> to be negated, since the NOT <key word> from <boolean factor> can precede a <between predicate>.

5.13 <in predicate>

Function

Specify a quantified comparison.

Format

<in predicate> ::=
{ IS_IN | NOT_IN } (<value expression> , { <subquery> | <in value list> })

<in value list> ::=
<value specification> { or <value specification> } ...

Effective Ada Declarations

For a program data type ct:

```
type IN_VALUE_LIST_ct is private;

function IS_IN ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function IS_IN ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : IN_VALUE_LIST_ct )
 return SEARCH_CONDITION;

function IS_IN ( LEFT : ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function IS_IN ( LEFT : ct ; RIGHT : IN_VALUE_LIST_ct )
 return SEARCH_CONDITION;

function NOT_IN ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function NOT_IN ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : IN_VALUE_LIST_ct )
 return SEARCH_CONDITION;

function NOT_IN ( LEFT : ct ; RIGHT : SUBQUERY_ct )
 return SEARCH_CONDITION;

function NOT_IN ( LEFT : ct ; RIGHT : IN_VALUE_LIST_ct )
 return SEARCH_CONDITION;

function "or"
         ( LEFT  : VALUE_SPECIFICATION_ct;
           RIGHT : VALUE_SPECIFICATION_ct ) return IN_VALUE_LIST_ct;

function "or" ( LEFT : VALUE_SPECIFICATION_ct ; RIGHT : ct )
 return IN_VALUE_LIST_ct;
```

```
function "or" ( LEFT : ct ; RIGHT : VALUE_SPECIFICATION_ct )
 return IN_VALUE_LIST_ct;

function "or" ( LEFT : ct ; RIGHT : ct ) return IN_VALUE_LIST_ct;

function "or" ( LEFT : IN_VALUE_LIST_ct ; RIGHT : VALUE_SPECIFICATION_ct )
 return IN_VALUE_LIST_ct;

function "or" ( LEFT : IN_VALUE_LIST_ct ; RIGHT : ct )
 return IN_VALUE_LIST_ct;
```

Example

```
PRIMARY_MANAGER,
ALTERNATE_MANAGER : EMPLOYEE_NAME;
CURSOR            : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => IS_IN ( MANAGER , PRIMARY_MANAGER or ALTERNATE_MANAGER ) ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => NOT_IN ( MANAGER ,
    SELEC     ( MANAGER,
    FROM     => EMPLOYEE,
    GROUP_BY => MANAGER,
    HAVING   => AVG ( SALARY ) > 20_000.00 ) ) ) );
```

Syntax Rules

1) The data types of the first <value expression> and the <subquery> or all <value specification>s in the <in value list> shall be the same.

2) If an <in value list> is used with boolean <value specification>s, then at least one of the first two <value specification>s in the <in value list> shall contain an <indicator specification>.

General Rules

1) Let x denote the result of the <value expression>. Let S denote the result of the <subquery> as in a <quantified predicate>, or the values specified by the <in value list>.

2) "IS_IN (x , S)" has the same result as "EQ (x , ANY (S))". "NOT_IN (x, S)" has the same result as "NOT IS_IN (x, S)".

Notes

1) There are four IS_IN (similarly, NOT_IN) functions effectively declared for each program data type. The functions differ in the types of the LEFT and RIGHT parameters, based on the text of the <in predicate> as follows:

   LEFT of type VALUE_EXPRESSION_ct (typed according to program type) - used when the <value expression> contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   LEFT of type ct (program type) - used when the <value expression> does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   RIGHT of type SUBQUERY_ct (typed according to program type) - used when a <subquery> is specified

   RIGHT of type IN_VALUE_LIST_ct (typed according to program type) - used when an <in value list> is specified

2) The overloaded "or" operators have three different types of operands. The effective type of each operand is based on the text of the corresponding <value specification> in the <in value list>, as follows:

   IN_VALUE_LIST_ct (typed according to program type) - always used for the left operand of the second and succeeding "or"s in an <in value list>; preceding "or" operators, which are applied from left to right in accordance with Ada rules, effectively return a result of type IN_VALUE_LIST_ct to be used as the left operand of the next "or". The remaining two types may be used for either operand of the first "or" in an <in value list>, and for the right operand of all succeeding "or"s.

   VALUE_SPECIFICATION_ct (typed according to program type) - used when the <value specification> contains an <indicator specification> or the <key word> USER

   ct (program type) - used when the <value specification> contains neither an <indicator specification> nor the <key word> USER

3) The Ada/SQL <in predicate> conforms to the ANSI SQL <in predicate>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	4
—	SR2	5
GR1-GR2	GR1-GR2	-

4) Ada/SQL SR1 expresses one aspect of Ada/SQL's strong typing.

5) A boolean <value specification> not containing an <indicator specification> is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If two such <value specification>s of a boolean program type were to be used as the first two <value specification>s within an <in value list>, then it would not be clear whether the "or" separating the <value specification>s should be taken as the predefined boolean "or" or as the Ada/SQL <value specification> separator. Furthermore, the effective Ada declarations could not be defined consistently with such ambiguity. By requiring an <indicator specification> in one of the first two <value specification>s in an <in value list>, SR2 ensures that this ambiguity is avoided. Note that the effective Ada declarations are such that a <value specification> for program data type ct, containing an <indicator specification>, when used within an <in value list>, effectively returns an object of type VALUE_SPECIFICATION_ct. Thus, the effective Ada/SQL "or" operator with at least one parameter of type VALUE_SPECIFICATION_ct is distinct from the predefined "or" operator with parameters of data type ct. The restriction of Ada/SQL SR2 should actually have virtually no impact on Ada/SQL programmers, because the limited range of boolean values makes their use in an <in value list> extremely limited. If a programmer insists upon using boolean values in an <in value list>, then one of the <value specification>s can be expressed as an <indicator specification> to comply with Ada/SQL SR2. For example, the following violates SR2 (assume that B1 and B2 are program variables of the same boolean data type, and that x is a <value expression> of that data type):

```
. . . IS_IN ( x , B1 or B2 ) . . .
```

However, the following has the exact same effect, and is permitted:

```
. . . IS_IN ( x , INDICATOR ( B1 ) or B2 ) . . .
```

6) The original Ada/SQL definition permitted only a single <value specification> in an <in value list>. ANSI SQL requires at least two <value specification>s in an <in value list>, and Ada/SQL now has the same requirement. A program executing an Ada/SQL statement with a single <value specification> in an <in value list> is erroneous with Release 1 implementations; later implementations will explicitly check for this situation.

5.14 <like predicate>

Function

Specify a pattern-match comparison.

Format

<like predicate> ::=
LIKE (<column specification> , <pattern>
[, ESCAPE => <escape character>])

<pattern> ::=
<value specification>

<escape character> ::=
<value specification>

Effective Ada Declarations

For a character string program data type ct, with components of data type cct:

```
function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : VALUE_SPECIFICATION_ct ) return SEARCH_CONDITION;

function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : ct ) return SEARCH_CONDITION;

function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : VALUE_SPECIFICATION_ct;
           ESCAPE  : ct ) return SEARCH_CONDITION;

function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : ct;
           ESCAPE  : ct ) return SEARCH_CONDITION;

function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : VALUE_SPECIFICATION_ct;
           ESCAPE  : cct ) return SEARCH_CONDITION;

function LIKE
         ( COLUMN  : COLUMN_SPECIFICATION_ct;
           PATTERN : ct;
           ESCAPE  : cct ) return SEARCH_CONDITION;
```

Example

```
LAST_NAME : EMPLOYEE_NAME; -- presumably set to, for example, "%Smith"
CURSOR    : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => LIKE ( NAME , LAST_NAME ) ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => LIKE ( NAME , LAST_NAME , ESCAPE => "^" ) ) );
                                    -- variation: '^'
```

Syntax Rules

1) The <column specification> shall reference a character string column.

2) The data type of the <pattern> shall be the same as the data type of the <column specification>.

3) The data type of the <escape character> shall be either: that of the <column specification>, or that of (character) components of the <column specification>.

4) The <value specification> of the <escape character> shall not contain an <indicator specification> or the <key word> USER.

General Rules

1) If the data type of the <escape character> is the same as the data type of the <column specification>, then the value of the <escape character> shall be a character string of length 1; otherwise, the DATA_EXCEPTION exception is raised.

2) Let x denote the value referenced by the <column specification> and let y denote the result of the <value specification> of the <pattern>.

3) If y is not the null value, then:

   Case:

   a) If an <escape character> is specified, then:

      i) Let z denote the result of the <value specification> of the <escape character>.

ii) There shall be a partitioning of the string y into substrings such that each substring is of length 1 or 2, no substring of length 1 is the escape character z, and each substring of length 2 is the escape character z followed by either the escape character z, an underscore character, or the percent sign character; otherwise, the DATA_EXCEPTION exception is raised.

In that partitioning of y, each substring of length 2 represents a single occurrence of the second character of that substring. Each substring of length 1 that is the underscore character represents an arbitrary character specifier. Each substring of length 1 that is the percent sign character represents an arbitrary string specifier. Each substring of length 1 that is neither the underscore character nor the percent sign character represents the character that it contains.

b) If an <escape character> is not specified, then each underscore character in y represents an arbitrary character specifier, each percent sign character in y represents an arbitrary string specifier, and each character in y that is neither the underscore character nor the percent sign character represents itself.

4) If y is not the null value, then the string y is a sequence of the minimum number of substring specifiers such that each <character> of y is part of exactly one substring specifier. A substring specifier is an arbitrary character specifier, an arbitrary string specifier, or any sequence of <character>s other than an arbitrary character specifier or an arbitrary string specifier.

5) "LIKE (x , y)" is unknown if x or y is the null value. If x and y are nonnull values, then "LIKE (x , y)" is either true or false.

6) "LIKE (x , y)" is true if there exists a partitioning of x into substrings such that:

a) A substring of x is a sequence of zero or more contiguous <character>s of x and each <character> of x is part of exactly one substring.

b) If the i-th substring specifier of y is an arbitrary character specifier, the i-th substring of x is any single <character>.

c) If the i-th substring specifier of y is an arbitrary string specifier, the i-th substring of x is any sequence of zero or more <character>s.

d) If the i-th substring specifier of y is neither an arbitrary character specifier nor an arbitrary string specifier, the i-th substring of x is equal to that substring specifier and has the same length as that substring specifier.

e) The number of substrings of x is equal to the number of substring specifiers of y.

Notes

1) There are six LIKE functions effectively declared for each character string program data type. The functions differ in the type of the PATTERN parameter and the type and existence of the ESCAPE

parameter, based on the text of the <like predicate> as follows:

PATTERN of type VALUE_SPECIFICATION_ct (typed according to program type) - used when the <pattern> <value specification> contains an <indicator specification> or the <key word> USER

PATTERN of type ct (program type) - used when the <pattern> <value specification> contains neither an <indicator specification> nor the <key word> USER

ESCAPE not present - used when the optional <escape character> is not specified

ESCAPE of type ct (program character string type) - used when the optional <escape character> is specified as a character string of the same data type as the <column specification>

ESCAPE of type cct (program character type) - used when the optional <escape character> is specified as a character of the same data type as the components of the <column specification>

2) The Ada/SQL <like predicate> conforms to the ANSI SQL <like predicate>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	-
SR2	SR2	3
SR3	SR3-SR4,GR1	4
GR1	GR2	-
GR2-GR3	GR3-GR4	5
GR4-GR5	GR5-GR6	-
GR6	—	6

3) Ada/SQL SR2 expresses one aspect of Ada/SQL's strong typing. Note that the data type of the <column specification> must be an unconstrained character string in order for the <pattern> to be of the same data type but of different (presumably shorter) length. The subtype of the <column specification> may be constrained, and may, of course, be different from the subtype of the <pattern>.

4) The original definition of Ada/SQL required that the data type of the <escape character> be the same as the data type of the <column specification>, and provision for this is retained for upward compatibility. Since the <escape character> is required to be only a single character, it is now also permitted to be of the same data type as the components of the <column specification>.

Ada/SQL SR3 expresses one aspect of Ada/SQL's strong typing. The length restriction of ANSI SQL SR3, the fact that the <escape character> must be a single character, is expressed in Ada/SQL GR1, since Ada constraint checking is performed at runtime.

Although ANSI SQL places no syntax restrictions on the <escape character>, the implication is

that it cannot be null. Ada/SQL SR4 explicitly forbids the <escape character> from being null, by not allowing its <value specification> to include an <indicator specification>. Use of the <key word> USER as the <escape character>, although syntactically permitted by <value specification>, would be meaningless, and so is explicitly prohibited by Ada/SQL SR4.

Release 1 implementations do not support specification of an <escape character> within a <like predicate>.

5) The Ada/SQL GRs include the qualification "if y is not the null value", because it is not at all clear how the ANSI SQL GRs apply if y is the null value, although ANSI SQL GR4 clearly contemplates the possibility.

6) Ada/SQL does not permit the NOT <key word> in a <like predicate>. Ada/SQL syntax does, however, permit the sense of a <like predicate> to be negated, since the NOT <key word> from <boolean factor> can precede a <like predicate>.

5.15 <null predicate>

Function

Specify a test for a null value.

Format

<null predicate> ::=
{ IS_NULL | IS_NOT_NULL } (<column specification>)

Effective Ada Declarations

```
function IS_NULL ( LEFT : COLUMN_SPECIFICATION ) return SEARCH_CONDITION;

function IS_NOT_NULL ( LEFT : COLUMN_SPECIFICATION )
 return SEARCH_CONDITION;
```

Example

```
CURSOR : CURSOR_NAME;
  . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => IS_NULL ( MANAGER ) ) ); -- variation: IS_NOT_NULL
```

Syntax Rules

None.

General Rules

1) Let x denote the value referenced by the <column specification>.

2) "IS_NULL (x)" is either true or false.

3) "IS_NULL (x)" is true if and only if x is the null value.

4) "IS_NOT_NULL (x)" has the same result as "NOT IS_NULL (x)".

Notes

1) The Ada/SQL <null predicate> conforms to the ANSI SQL <null predicate>.

2) Release 1 implementations do not support the <null predicate>.

5.16 <quantified predicate>

Function

Specify a quantified comparison.

Format

<quantified predicate> ::=
  <equality operator> (<value expression> , <quantifier> (<subquery>))
 | <value expression> <ordering operator> <quantifier> (<subquery>)

<quantifier> ::=
  <all> | <some>

<all> ::= ALLL

<some> ::= SOME | ANY

Effective Ada Declarations

For a program data type ct:

```
type QUANTIFIER_ct is private;

function ALLL ( LEFT : SUBQUERY_ct ) return QUANTIFIER_ct;

function SOME ( LEFT : SUBQUERY_ct ) return QUANTIFIER_ct;

function ANY ( LEFT : SUBQUERY_ct ) return QUANTIFIER_ct;

function EQ ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function EQ ( LEFT : ct ; RIGHT : QUANTIFIER_ct ) return SEARCH_CONDITION;

function NE ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function NE ( LEFT : ct ; RIGHT : QUANTIFIER_ct ) return SEARCH_CONDITION;

function "<" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function "<" ( LEFT : ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function ">" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;
```

```
function ">" ( LEFT : ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function "<=" ( LEFT : ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function ">=" ( LEFT : VALUE_EXPRESSION_ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;

function ">=" ( LEFT : ct ; RIGHT : QUANTIFIER_ct )
 return SEARCH_CONDITION;
```

Example

```
package E is new EMPLOYEE_CORRELATION.NAME ( "E" );

CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => E.EMPLOYEE,
  WHERE => SALARY >= ALLL (
    SELEC  ( SALARY,
    FROM   => EMPLOYEE,
    WHERE => EQ ( MANAGER , E.MANAGER ) ) ) ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE => EQ ( NAME , ANY (    -- variation: SOME
    SELEC ( MANAGER,
    FROM => EMPLOYEE ) ) ) ) );
```

Syntax Rules

1) The data types of the <value expression> and the <subquery> shall be the same.

General Rules

1) Let x denote the result of the <value expression> and let S denote the result of the <subquery>.

2) The result of the <quantified predicate> "<equality operator> (x , <quantifier> (S))" or "x <ordering operator> <quantifier> (S)" is derived by the application of the implied <comparison predicate> "<equality operator> (x , s)" or "x <ordering operator> s" to every value in S:

   Case:

a) If S is empty or if the implied <comparison predicate> is true for every value s in S, then the <quantified predicate> with the <all> <quantifier> is true.

b) If the implied <comparison predicate> is false for at least one value s in S, then the <quantified predicate> with the <all> <qualifier> is false.

c) If the implied <comparison predicate> is true for at least one value s in S, then the <quantified predicate> with the <some> <qualifier> is true.

d) If S is empty or if the implied <comparison predicate> is false for every value s in S, then the <quantified predicate> with the <some> <qualifier> is false.

e) If the <quantified predicate> is neither true nor false, then it is unknown.

Notes

1) There are two functions effectively declared for each operator that may be used in a <quantified predicate>. The functions differ in the type of their first parameter, based on the text of the <value expression> within the <quantified predicate> as follows:

   VALUE_EXPRESSION_ct (typed according to program type) - used when the <value expression> contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   ct (program type) - used when the <value expression> does not contain any of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

2) The Ada/SQL operators corresponding to ANSI SQL "=" and "<>" are written as prefix EQ and NE, due to Ada restrictions on overloading infix "=" and "/=".

3) The Ada/SQL <quantified predicate> conforms to the ANSI SQL <quantified predicate>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	4
GR1-GR2	GR1-GR2	-

4) SR1 expresses one aspect of Ada/SQL's strong typing.

5) Release 1 implementations do not support the <quantified predicate>.

5.17 <exists predicate>

Function

Specify a test for an empty set.

Format

<exists predicate> ::=
EXISTS (<subquery>)

Effective Ada Declarations

```
function EXISTS ( LEFT : SUBQUERY ) return SEARCH_CONDITION;
```

Example

```
package E is new EMPLOYEE_CORRELATION.NAME ( "E" );

CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => E.EMPLOYEE,
  WHERE => EXISTS (
    SELEC  ( '*',
    FROM  => EMPLOYEE,
    WHERE => EQ ( MANAGER , E.NAME ) ) ) ) );
```

Syntax Rules

None.

General Rules

1) Let S denote the result of the <subquery>.

2) "EXISTS (S)" is either true or false.

3) "EXISTS (S)" is true if and only if S is not empty.

Notes

1) The Ada/SQL <exists predicate> conforms to the ANSI SQL <exists predicate>.

2) Release 1 implementations do not support the <exists predicate>.

5.18 <search condition>

Function

Specify a condition that is "true", "false", or "unknown" depending on the result of applying boolean operators to specified conditions.

Format

<search condition> ::=
  <boolean factor> [{ AND <boolean factor> } ...]
 | <boolean factor> [{ OR <boolean factor> } ...]

<boolean factor> ::=
  [NOT] <boolean primary>

<boolean primary> ::=
  <predicate> | (<search condition>)

Effective Ada Declarations

```
type SEARCH_CONDITION is private;

NULL_SEARCH_CONDITION : constant SEARCH_CONDITION;

function "AND" ( LEFT, RIGHT : SEARCH_CONDITION ) return SEARCH_CONDITION;

function "OR" ( LEFT, RIGHT : SEARCH_CONDITION ) return SEARCH_CONDITION;

function "NOT" ( LEFT : SEARCH_CONDITION ) return SEARCH_CONDITION;
```

Example

```
PRIMARY_MANAGER,
ALTERNATE_MANAGER : EMPLOYEE_NAME;
CURSOR            : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => NOT BETWEEN ( SALARY , 20_000.00 AND 30_000.00 )
  AND      ( EQ ( MANAGER , PRIMARY_MANAGER )
  OR         EQ ( MANAGER , ALTERNATE_MANAGER ) ) ) );
```

Syntax Rules

1) A <column specification> or <value expression> specified in a <search condition> is directly contained in that <search condition> if the <column specification> or <value expression> is not specified within a <set function specification> or a <subquery> of that <search condition>.

General Rules

1) The result is derived by the application of the specified boolean operators to the conditions that result from the application of each specified <predicate> to a given row of a table or a given group of a grouped table. If boolean operators are not specified, then the result of the <search condition> is the result of the specified <predicate>.

2) NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown. AND and OR are defined by the following truth tables:

AND	*true*	*false*	*unknown*
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

OR	*true*	*false*	*unknown*
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

3) Expressions within parentheses are evaluated first and when the order of evaluation is not specified by parentheses, NOT is applied before AND and OR, and ANDs and ORs are applied from left to right. (ANDs and ORs may not be intermixed without using parentheses to clearly show precedence.)

4) When a <search condition> is applied to a row of a table, each reference to a column of that table by a <column specification> directly contained in the <search condition> is a reference to the value of that column in that row.

Notes

1) Release 1 implementations do not support null values. Hence, "unknown" conditions cannot be created.

2) The Ada/SQL <search condition> conforms to the ANSI SQL <search condition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	-
GR1-GR2	GR1-GR2	-
GR3	GR3	3
GR4	GR4	-

3) ANSI SQL provides precedence of AND over OR in <search condition>s. Ada, and hence Ada/SQL, requires that expressions containing both ANDs and ORs be parenthesized to clearly show the order of evaluation.

5.19 <table expression>

Function

Specify a table or a grouped table.

Format

<table expression> ::=
  <from clause>
 [, <where clause>]
 [, <group by clause>]
 [, <having clause>]

Effective Ada Declarations

see sections declaring FROM, WHERE, GROUP_BY, and HAVING <key word>s

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => SALARY > 25_000.00 ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC     ( MANAGER & AVG ( SALARY ),
  FROM     => EMPLOYEE,
  GROUP_BY => MANAGER,
  HAVING   => AVG ( SALARY ) > 25_000.00 ) );
```

Syntax Rules

1) If the table identified in the <from clause> is a grouped view, then the <table expression> shall not contain a <where clause>, <group by clause>, or <having clause>.

General Rules

1) If all optional clauses are omitted, then the table is the result of the <from clause>. Otherwise, each specified clause is applied to the result of the previously specified clause and the table is the result of the application of the last specified clause. The result of a <table expression> is a derived table in which the i-th column inherits the description of the i-th column of the table specified by the <from clause>.

Notes

1) The Ada/SQL <table expression> conforms to the ANSI SQL <table expression>.

5.20 <from clause>

Function

Specify a table derived from one or more named tables.

Format

<from clause> ::=
FROM => <table reference> [{ & <table reference> } ...]

<table reference> ::=
[<correlation name> .] <table name>

Effective Ada Declarations

```
type FROM_CLAUSE is private;

function "&" ( LEFT, RIGHT : FROM_CLAUSE ) return FROM_CLAUSE;

For a table t with <authorization identifier> a:

  Within generic packages t_CORRELATION.NAME and a_t_CORRELATION.NAME:

    function t return FROM_CLAUSE;

    type TABLE_REFERENCE is
      record
        t : FROM_CLAUSE;
      end record;

    function a return TABLE_REFERENCE;
```

see also sections declaring FROM <key word>

Example

```
package E is new EMPLOYEE_CORRELATION.NAME ( "E" ); -- employees
package M is new EMPLOYEE_CORRELATION.NAME ( "M" ); -- managers

CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => SALARY > 25_000.00 ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( E.NAME & E.SALARY & E.MANAGER,
  FROM   => E.EMPLOYEE & M.EMPLOYEE,
```

```
WHERE => EQ ( E.MANAGER , M.NAME )
AND      E.SALARY > M.SALARY ) );
```

Syntax Rules

1) A <table name> specified in a <table reference> is exposed in the containing <from clause> if and only if that <table reference> does not specify a <correlation name>.

2) A <table name> that is exposed in a <from clause> shall not be the same as any other <table name> that is exposed in that <from clause>.

3) A <correlation name> specified in a <table reference> shall not be the same as any other <correlation name> specified in the containing <from clause>, and shall not be the same as the <table identifier> of any <table name> that is exposed in the containing <from clause>.

4) The scope of <correlation name>s and exposed <table name>s specified in a <from clause> is the innermost <subquery>, <query specification>, or <select statement> that contains the <table expression> in which the <from clause> is contained. A <table name> that is specified in a <from clause> has a scope defined by that <from clause> if and only if the <table name> is exposed in that <from clause>.

5) If the table identified by <table name> is a grouped view, then the <from clause> shall contain exactly one <table reference>.

6) Case:

   a) If the <from clause> contains a single <table name>, then the description of the result of the <from clause> is the same as the description of the table identified by that <table name>.

   b) If the <from clause> contains more than one <table name>, then the description of the result of the <from clause> is the concatenation of the descriptions of the tables identified by those <table name>s, in the order in which the <table name>s appear in the <from clause>.

7) If a <table reference> contains a <correlation name>, then that <correlation name> shall have been declared for the table denoted by the <table name>, in exactly one of the <global variable package>s or <local variable package>s that apply to the <Ada/SQL compilation unit> containing the <table reference>.

General Rules

1) The specification of a <correlation name> or exposed <table name> in a <table reference> defines that <correlation name> or <table name> as a designator of the table identified by the <table name> of that <table reference>.

2) Case:

   a) If the <from clause> contains a single <table name>, then the result of the <from clause> is the table identified by that <table name>.

   b) If the <from clause> contains more than one <table name>, then the result of the <from clause> is the extended Cartesian product of the tables identified by those <table name>s. The extended Cartesian product, R, is the multi-set of all rows r such that r is the concatenation of a row from each of the identified tables in the order in which they are identified. The cardinality of R is the product of the cardinalities of the identified tables. The ordinal position of a column in R is n+s, where n is the ordinal position of that column in the named table T from which it is derived and s is the sum of the degrees of the tables identified before T in the <from clause>.

Notes

1) The effective Ada declarations for functions a and t are used with <correlation name>s. Functions returning FROM_CLAUSE are effectively declared for <table name>s (see 5.4), for use without <correlation name>s.

   The generic packages t_CORRELATION.NAME and a_t_CORRELATION.NAME are used to declare <correlation name>s for table a.t. For example, the <correlation name> cn may be declared as:

```
package cn is new t_CORRELATION.NAME ( "cn" ); or

package cn is new a_t_CORRELATION.NAME ( "cn" );
```

   The <table reference> cn.t effectively calls function t, declared within the generic package, to return the appropriate FROM_CLAUSE value.

   Likewise, the <table reference> cn.a.t effectively calls function a, declared within the generic package, to return a TABLE_REFERENCE value, then selects the t component of this value, which is the appropriate FROM_CLAUSE value.

   Note that the <correlation name> precedes the <table name> in an Ada/SQL <table reference>; this is the reverse of ANSI SQL.

   Release 1 implementations do not support <authorization identifier>s within <table name>s . Hence, only generic package t_CORRELATION.NAME is available for table t, and function a is not available within it.

2) The Ada/SQL <from clause> conforms to the ANSI SQL <from clause>.

5.21 <where clause>

Function

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

Format

<where clause> ::=
WHERE => <search condition>

Effective Ada Declarations

see sections declaring WHERE <key word>

Example

```
CURSOR : CURSOR_NAME;
  . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => SALARY > 25_000.00 ) );
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause>. Each <column specification> directly contained in the <search condition> shall unambiguously reference a column of T or be an outer reference.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

2) A <value expression> directly contained in the <search condition> shall not include a reference to a column derived from a function.

3) If a <value expression> directly contained in the <search condition> is a <set function specification>, then the <where clause> shall be contained in a <having clause> and the <column specification> in the <set function specification> shall be an outer reference.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

General Rules

1) Let R denote the result of the <from clause>.

2) The <search condition> is applied to each row of R. The result of the <where clause> is a table of those rows of R for which the result of the <search condition> is true.

3) Each <subquery> in the <search condition> is effectively executed for each row of R and the results used in the application of the <search condition> to the given row of R. If any executed <subquery> contains an outer reference to a column of R, then the reference is to the value of that column in the given row of R.

NOTE: "Outer reference" is defined in 5.7, "<column specification>".

Notes

1) The Ada/SQL <where clause> conforms to the ANSI SQL <where clause>.

5.22 <group by clause>

Function

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

Format

<group by clause> ::=
GROUP_BY => <column specification> [{ & <column specification> } ...]

Effective Ada Declarations

```
type GROUP_BY_CLAUSE is private;

NULL_GROUP_BY_CLAUSE : constant GROUP_BY_CLAUSE;

function "&" ( LEFT : GROUP_BY_CLAUSE ; RIGHT : GROUP_BY_CLAUSE )
 return GROUP_BY_CLAUSE;
```

see sections declaring GROUP_BY <key word>

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => NOT_IN ( MANAGER ,
    SELEC     ( MANAGER,
    FROM     => EMPLOYEE,
    GROUP_BY => MANAGER,
    HAVING   => AVG ( SALARY ) > 20_000.00 ) ) ) );
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC     ( MANAGER & SALARY & COUNT ( '*' ),
  FROM     => EMPLOYEE,
  GROUP_BY => MANAGER & SALARY ) );
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause> or <where clause>.

2) Each <column specification> in the <group by clause> shall unambiguously reference a column of T. A column referenced in a <group by clause> is a grouping column.

General Rules

1) Let R denote the result of the preceding <from clause> or <where clause>.

2) The result of the <group by clause> is a partitioning of R into a set of groups. The set is the minimum number of groups such that, for each grouping column of each group of more than one row, all values of that grouping column are identical.

3) Every row of a given group contains the same value of a given grouping column. When a <search condition> or <value expression> is applied to a group, a reference to a grouping column is a reference to that value.

Notes

1) Functions returning GROUP_BY_CLAUSE are effectively declared for <column specification> (see 5.7).

2) The Ada/SQL <group by clause> conforms to the ANSI SQL <group by clause>.

5.23 <having clause>

Function

Specify a restriction on the grouped table resulting from the previous <group by clause> or <from clause> by eliminating groups not meeting the <search condition>.

Format

<having clause> ::=
HAVING => <search condition>

Effective Ada Declarations

see sections declaring HAVING <key word>

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => EMPLOYEE,
  WHERE  => NOT_IN ( MANAGER ,
    SELEC      ( MANAGER,
    FROM       => EMPLOYEE,
    GROUP_BY   => MANAGER,
    HAVING     => AVG ( SALARY ) > 20_000.00 ) ) ) );
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause>, <where clause>, or <group by clause>. Each <column specification> directly contained in the <search condition> shall unambiguously reference a grouping column of T or be an outer reference.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

2) Each <column specification> contained in a <subquery> in the <search condition> that references a column of T shall reference a grouping column of T or shall be specified within a <set function specification>.

General Rules

1) Let R denote the result of the preceding <from clause>, <where clause>, or <group by clause>. If that clause is not a <group by clause>, then R consists of a single group and does not have a grouping column.

2) The <search condition> is applied to each group of R. The result of the <having clause> is a grouped table of those groups of R for which the result of the <search condition> is true.

3) When the <search condition> is applied to a given group of R, that group is the argument or argument source of each <set function specification> directly contained in the <search condition> unless the <column specification> in the <set function specification> is an outer reference.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

4) Each <subquery> in the <search condition> is effectively executed for each group of R and the result used in the application of the <search condition> to the given group of R. If any executed <subquery> contains an outer reference to a column of R, then the reference is to the values of that column in the given group of R.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

Notes

1) The Ada/SQL <having clause> conforms to the ANSI SQL <having clause>.

5.24 <subquery>

Function

Specify a multi-set of values derived from the result of a <table expression>.

Format

<subquery> ::
[SELEC | SELECT_ALL | SELECT_DISTINCT | SELEC_ALL | SELEC_DISTINCT]
(<result specification> ,
<table expression>)

<result specification> ::=
<value expression>
| '*'

Effective Ada Declarations

```
-- see 8.10 for declaration of type STAR_TYPE

type SUBQUERY is private ;

function SELEC
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELEC
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_ALL
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_ALL
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
```

```
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_DISTINCT
          ( WHAT     : VALUE_EXPRESSION;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_DISTINCT
          ( WHAT     : STAR_TYPE;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELEC_ALL
          ( WHAT     : VALUE_EXPRESSION;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_ALL;

function SELEC_ALL
          ( WHAT     : STAR_TYPE;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_ALL;

function SELEC_DISTINCT
          ( WHAT     : VALUE_EXPRESSION;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_DISTINCT;

function SELEC_DISTINCT
          ( WHAT     : STAR_TYPE;
            FROM     : FROM_CLAUSE;
            WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_DISTINCT;
```

For a program data type ct:

```
type SUBQUERY_ct is private;

function SELEC
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELEC
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELEC
         ( WHAT     : VALUE_EXPRESSION_ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELEC
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELECT_ALL
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_ALL
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;
```

```
function SELECT_ALL
         ( WHAT      : VALUE_EXPRESSION_ct;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELECT_ALL
         ( WHAT      : STAR_TYPE;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELECT_DISTINCT
         ( WHAT      : ct;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY;

function SELECT_DISTINCT
         ( WHAT      : ct;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELECT_DISTINCT
         ( WHAT      : VALUE_EXPRESSION_ct;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELECT_DISTINCT
         ( WHAT      : STAR_TYPE;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY  : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING    : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct;

function SELEC_ALL
         ( WHAT      : ct;
           FROM      : FROM_CLAUSE;
           WHERE     : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
```

```
            GROUP_BY : GROUP_BY_CLAUSE   := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_ALL;

function SELEC_ALL
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_ALL;

function SELEC_ALL
         ( WHAT     : VALUE_EXPRESSION_ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_ALL;

function SELEC_ALL
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_ALL;

function SELEC_DISTINCT
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY renames SELECT_DISTINCT;

function SELEC_DISTINCT
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_DISTINCT;

function SELEC_DISTINCT
         ( WHAT     : VALUE_EXPRESSION_ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_DISTINCT;
```

```
function SELEC_DISTINCT
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return SUBQUERY_ct renames SELECT_DISTINCT;
```

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => SALARY >
    SELEC ( AVG ( SALARY ),      -- variations: SELECT_ALL, SELECT_DISTINCT,
    FROM => EMPLOYEE ) ) );      --             SELEC_ALL, SELEC_DISTINCT
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => EMPLOYEE,
  WHERE => IS_IN ( NAME ,
    SELEC ( '*',                 -- variations: SELECT_ALL, SELECT_DISTINCT,
    FROM => MANAGERS ) ) ) );    --             SELEC_ALL, SELEC_DISTINCT
```

– assume MANAGERS is a database table with one column, containing the names
– of all managers

Syntax Rules

1) Specifying SELEC_ALL is equivalent to specifying SELECT_ALL; specifying SELEC_DISTINCT is equivalent to specifying SELECT_DISTINCT.

2) The applicable <privileges> for each <table name> contained in the <table expression> shall include SELEC.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

3) Case:

   a) If the <result specification> '*' is specified in a <subquery> of any <predicate> other than an <exists predicate>, then the degree of the <table expression> shall be 1, and the <result specification> is equivalent to a <value expression> consisting of a <column specification> that references the sole column of the <table expression>.

   b) If the <result specification> '*' is specified in a <subquery> of an <exists predicate>, then the <result specification> is equivalent to an arbitrary <value expression> that does not

include a <set function specification> and that is allowed in the <subquery>.

4) The data type of the values of the <subquery> is the data type of the implicit or explicit <value expression>.

5) Let R denote the result of the <table expression>.

6) Each <column specification> in the <value expression> shall unambiguously reference a column of R.

7) If R is a grouped view, then the <result specification> shall not contain a <set function specification>.

8) If R is a grouped table, then each <column specification> in the <value expression> shall reference a grouping column or be specified within a <set function specification>. If R is not a grouped table and the <value expression> includes a <set function specification>, then each <column specification> in the <value expression> shall be specified within a <set function specification>.

9) The _DISTINCT suffix shall not be specified on more than one <key word> in a <subquery>, excluding any <subquery> contained in that <subquery>.

10) If a <subquery> is specified in a <comparison predicate>, then the <table expression> shall not contain a <group by clause> or a <having clause> and shall not identify a grouped view.

General Rules

1) If R is not a grouped table and the <value expression> includes a <set function specification>, then R is the argument or argument source of each <set function specification> in the <value expression> and the result of the <subquery> is the value specified by the <value expression>.

2) If R is not a grouped table and the <value expression> does not include a <set function specification>, then the <value expression> is applied to each row of R yielding a multi-set of n values, where n is the cardinality of R. If neither SELECT_DISTINCT nor SELEC_DISTINCT is specified, then the multi-set is the result of the <subquery>. If SELECT_DISTINCT or SELEC_DISTINCT is specified, then the result of the <subquery> is the set of values derived from that multi-set by the elimination of any redundant duplicate values.

3) If R is a grouped table, then the <value expression> is applied to each group of R yielding a multi-set of n values, where n is the number of groups in R. When the <value expression> is applied to a given group of R, that group is the argument or argument source of each <set function specification> in the <value expression>. If neither SELECT_DISTINCT nor SELEC_DISTINCT is specified, then the multi-set is the result of the <subquery>. If SELECT_DISTINCT or SELEC_DISTINCT is specified, then the result of the <subquery> is the set of values derived from that multi-set by the elimination of any redundant duplicate values.

Notes

1) There are six <subquery> functions effectively declared for each <key word> that may be used to introduce the construct. The functions differ in the type of the their first parameter and their return type, based on the text and context of the <subquery> as follows:

   Return type SUBQUERY - used in contexts where the data type of the <subquery> is not important: <exists predicate>

   Return type SUBQUERY_ct (typed according to program type) - used in contexts where strong typing is applied to the <subquery>: <comparison predicate>, <in predicate>, <quantified predicate>

   Parameter type VALUE_EXPRESSION - used only with return type SUBQUERY; used when the <result specification> is a <value expression> containing at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   Parameter type VALUE_EXPRESSION_ct (typed according to program type) - used only with return type SUBQUERY_ct; used when the <result specification> is a <value expression> containing at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   Parameter type ct (program type) - used with either return type; used when the <result specification> is a <value expression> not containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

   Parameter type STAR_TYPE - used with either return type; used when the <result specification> is '*'

2) The Ada/SQL <subquery> conforms to the ANSI SQL <subquery>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	3
SR1-SR9	SR2-SR10	-
GR1-GR3	GR1-GR3	-

3) The <key word>s SELECT_ALL and SELECT_DISTINCT are those originally defined for Ada/SQL, and are provided for upward compatibility. SELEC_ALL and SELEC_DISTINCT are provided to use the same SELEC keyword as for <privileges> and because some users have expressed a preference for them. Release 1 implementations do not recognize the new <key word>s.

5.25 <query specification>

Function

Specify a table derived from the result of a <table expression>.

Format

<query specification> ::=
[SELEC | SELECT_ALL | SELECT_DISTINCT | SELEC_ALL | SELEC_DISTINCT]
(<select list> ,
<table expression>)

<select list> ::=
<value expression> [{ & <value expression> } ...]
| <value expression> [{ and <value expression> } ...]
| '*'

Effective Ada Declarations

```
-- see 8.10 for declaration of type STAR_TYPE

type QUERY_SPECIFICATION is private;

type SELECT_LIST is private;

function SELEC
         ( WHAT     : SELECT_LIST;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELEC
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELEC
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;
```

```
function SELECT_ALL
         ( WHAT     : SELECT_LIST;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_ALL
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_ALL
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_DISTINCT
         ( WHAT     : SELECT_LIST;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_DISTINCT
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_DISTINCT
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELEC_ALL
         ( WHAT     : SELECT_LIST;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
```

```
            GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
            HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_ALL;

function SELEC_ALL
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_ALL;

function SELEC_ALL
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_ALL;

function SELEC_DISTINCT
         ( WHAT     : SELECT_LIST;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_DISTINCT;

function SELEC_DISTINCT
         ( WHAT     : VALUE_EXPRESSION;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_DISTINCT;

function SELEC_DISTINCT
         ( WHAT     : STAR_TYPE;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_DISTINCT;

function "&" ( LEFT : VALUE_EXPRESSION ; RIGHT : VALUE_EXPRESSION )
 return SELECT_LIST;

function "&" ( LEFT : SELECT_LIST ; RIGHT : VALUE_EXPRESSION )
 return SELECT_LIST;

function "and" ( LEFT : VALUE_EXPRESSION ; RIGHT : VALUE_EXPRESSION )
 return SELECT_LIST renames "&";
```

```
function "and" ( LEFT : SELECT_LIST ; RIGHT : VALUE_EXPRESSION )
 return SELECT_LIST renames "&";
```

For a program data type ct:

```
function SELEC
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_ALL
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELECT_DISTINCT
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION;

function SELEC_ALL
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_ALL;

function SELEC_DISTINCT
         ( WHAT     : ct;
           FROM     : FROM_CLAUSE;
           WHERE    : SEARCH_CONDITION := NULL_SEARCH_CONDITION;
           GROUP_BY : GROUP_BY_CLAUSE  := NULL_GROUP_BY_CLAUSE;
           HAVING   : SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 return QUERY_SPECIFICATION renames SELECT_DISTINCT;

function "&" ( LEFT : VALUE_EXPRESSION ; RIGHT : ct ) return SELECT_LIST;

function "&" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION ) return SELECT_LIST;

function "&" ( LEFT : SELECT_LIST ; RIGHT : ct ) return SELECT_LIST;

function "and" ( LEFT : VALUE_EXPRESSION ; RIGHT : ct ) return SELECT_LIST
```

```
    renames "&";

  function "and" ( LEFT : ct ; RIGHT : VALUE_EXPRESSION ) return SELECT_LIST
    renames "&";

  function "and" ( LEFT : SELECT_LIST ; RIGHT : ct ) return SELECT_LIST
    renames "&";
```

Example

```
CURSOR : CURSOR_NAME;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC ( NAME & SALARY & MANAGER, -- variation: NAME and SALARY and MANAGER
  FROM => EMPLOYEE ) );
 . . .
INSERT_INTO ( EMPLOYEE ,
  SELEC ( '*',
  FROM => NEW_EMPLOYEE_FILE ) );

-- assume NEW_EMPLOYEE_FILE is another database table structured identically
-- to the EMPLOYEE table

-- variations in the above: replace SELEC with SELECT_ALL, SELECT_DISTINCT,
--  SELEC_ALL, or SELEC_DISTINCT
```

Syntax Rules

1) Specifying SELEC_ALL is equivalent to specifying SELECT_ALL; specifying SELEC_DISTINCT is equivalent to specifying SELECT_DISTINCT.

2) If a <select list> contains two or more <value expression>s, then at least one of the first two <value expression>s in the <select list> shall contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER.

3) Two <select list>s, differing only in the use of "&" vs. "and" to separate <value expression>s, are equivalent.

4) The applicable <privileges> for each <table name> contained in the <table expression> shall include SELEC.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, <privilege definition>".

5) Let R denote the result of the <table expression>.

6) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

7) The <select list> '*' is equivalent to a <value expression> sequence in which each <value expression> is a <column specification> that references a column of R and each column of R is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within R.

8) Each <column specification> in each <value expression> shall unambiguously reference a column of R. The _DISTINCT suffix shall not be specified on more than one <key word> in a <query specification>, excluding any <subquery> of that <query specification>.

9) If R is a grouped view, then the <select list> shall not contain a <set function specification>.

10) If R is a grouped table, then each <column specification> in each <value expression> shall reference a grouping column or be specified within a <set function specification>. If R is not a grouped table and any <value expression> includes a <set function specification>, then every <column specification> in every <value expression> shall be specified within a <set function specification>.

10) Each column of the table that is the result of a <query specification> has the same data type and length (for character strings) as the <value expression> from which the column was derived.

11) If the i-th <value expression> in the <select list> consists of a single <column specification>, then the i-th column of the result is a named column whose <column name> is that of the <column specification>. Otherwise, the i-th column is an unnamed column.

12) A column of the table that is the result of a <query specification> is constrained to contain only nonnull values if and only if it is a named column that is constrained to contain only nonnull values.

13) A <query specification> is updatable if and only if the following conditions hold:

   a) Neither SELECT_DISTINCT nor SELEC_DISTINCT is specified.

   b) Every <value expression> in the <select list> consists of a <column specification>, or a <column specification> to which one or more CONVERT_TO operators are applied.

   c) The <from clause> of the <table expression> specifies exactly one <table reference>, and that <table reference> refers to an updatable table.

   d) The <where clause> of the <table expression> does not include a <subquery>.

   e) The <table expression> does not include a <group by clause> or a <having clause>.

General Rules

1) If R is not a grouped table and the <select list> includes a <set function specification>, then R is the argument or argument source of each <set function specification> in the <select list> and the result of the <query specification> is a table consisting of one row. The i-th value of the row is the

value specified by the i-th <value expression>.

2) If R is not a grouped table and the <select list> does not include a <set function specification>, then each <value expression> is applied to each row of R yielding a table of m rows, where m is the cardinality of R. The i-th column of the table contains the values derived by the applications of the i-th <value expression>. If neither SELECT_DISTINCT nor SELEC_DISTINCT is specified, then the table is the result of the <query specification>. If SELECT_DISTINCT or SELEC_DISTINCT is specified, then the result of the <query specification> is the table derived from that table by the elimination of any redundant duplicate rows.

3) If R is a grouped table that has zero groups, then the result of the <query specification> is an empty table.

4) If R is a grouped table that has one or more groups, then each <value expression> is applied to each group of R yielding a table of m rows, where m is the number of groups in R. The i-th column of the table contains the values derived by the applications of the i-th <value expression>. When a <value expression> is applied to a given group of R, that group is the argument or argument source of each <set function specification> in the <value expression>. If neither SELECT_DISTINCT nor SELEC_DISTINCT is specified, then the table is the result of the <query specification>. If SELECT_DISTINCT or SELEC_DISTINCT is specified, then the result of the <query specification> is the table derived from that table by the elimination of any redundant duplicate rows.

5) A row is a duplicate of another row if and only if all pairs of values with the same ordinal position are identical.

Notes

1) There are four <query specification> functions effectively declared for each <key word> that may be used to introduce the construct. The functions differ in the type of their first parameter, based on the text of the <query specification> as follows:

SELECT_LIST - used when the <select list> contains more than one <value expression>

VALUE_EXPRESSION - used when the <select list> contains only one <value expression>, which contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

ct (program type) - used when the <select list> contains only one <value expression>, which does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

STAR_TYPE - used when the <select list> consists of the single element '*'

2) The overloaded "&" (equivalently, "and") operators have three different types of operands. The effective type of each operand is based on the text of the corresponding <value expression> in the <select list>, as follows:

SELECT_LIST - always used fc the left operand of the second and succeeding "&"s in a <select

list>; preceding "&" operators, which are applied from left to right in accordance with Ada rules, effectively return a result of type SELECT_LIST to be used as the left operand of the next "&". The remaining two types may be used for either operand of the first "&" in a <select list>, and for the right operand of all succeeding "&"s

VALUE_EXPRESSION - used when the <value expression> contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

ct (program type) - used when the <value expression> does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

3) The Ada/SQL <query specification> conforms to the ANSI SQL <query specification>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	4
—	SR2	5
—	SR3	6
SR1-SR7	SR4-SR10	-
SR8	SR11	7
SR9	SR12	-
SR10	SR13	8
SR11	SR14	-
GR1-GR2	GR1-GR2	-
GR3	GR3	9
GR4-GR5	GR4-GR5	-

4) The <key word>s SELECT_ALL and SELECT_DISTINCT are those originally defined for Ada/SQL, and are provided for upward compatibility. SELEC_ALL and SELEC_DISTINCT are provided to use the same SELEC keyword as for <privileges> and because some users have expressed a preference for them. Release 1 implementations do not recognize the new <key word>s.

5) A <value expression> not containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER is, as far as the effective Ada declarations are concerned, of a user-defined (or predefined) program type. If two such <value expression>s of program data types t1 and t2, were to be used as the first two <value expression>s within a <select list>, then the following "&" (or, equivalently, "and") operator must be effectively declared:

```
function "&" ( LEFT : t1 ; RIGHT : t2 ) return SELECT_LIST;
```

An implementation which generates all possible effective Ada/SQL declarations based on the type declarations contained in a <schema>, would have to generate n**2 such functions, where n is the number of different data types declared. By prohibiting the first two <value expression>s within a

<select list> from both being of program types, SR2 ensures that the number of functions effectively declared is linear in the number of program data types defined, rather than depending on the square of the number of data types. Note that the effective Ada/SQL declarations are such that a <value expression> containing a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER, when used within a <select list>, effectively returns an object of the single type VALUE_EXPRESSION, regardless of its underlying conceptual type, thereby avoiding the n**2 problem.

Release 1 implementations do not support the <key word> USER.

The restriction of Ada/SQL SR2 should actually have virtually no impact on Ada/SQL programmers, because the usefulness of a single <value expression> (let alone two!) not containing a <column specification>, a <set function specification>, or the <key word> USER is extremely limited in a <select list>. The value returned by such a <value expression> is not dependent on the database, and so could easily be computed outside of the Ada/SQL statement.

If a programmer insists on using such program expressions as the first two <value expression>s in a <select list>, then one of the <value expression>s can be expressed as an <indicator specification> to comply with Ada/SQL SR2. For example, the following violates SR2 (assume that NEW_EMPLOYEE and HIS_MANAGER are program variables of character string type EMPLOYEE_NAME):

```
SELEC ( NEW_EMPLOYEE & HIS_MANAGER . . .
```

However, the following has the exact same effect, and is permitted:

```
SELEC ( INDICATOR ( NEW_EMPLOYEE ) & HIS_MANAGER . . .
```

The above example also shows another potential problem avoided by SR2: The first two <value expression>s in the illegal SELEC fragment are of the same character string program data type, for which the "&" operator is already defined as catenation. (A similar problem exists for two boolean values with an "and" operator.) If NEW_EMPLOYEE and HIS_MANAGER are to be treated as separate columns of the result, rather than as a single catenated value, then the effective declaration for the Ada/SQL "&" operator must redefine the predefined "&" operator for the EMPLOYEE_NAME character string type. The ambiguity as to whether or not NEW_EMPLOYEE and HIS_MANAGER are to be treated as separate columns or as a single catenated value, as well as the undesirability of redefining the "&" operator for the former case, make it fortunate that Ada/SQL SR2 simply prohibits such a construct.

6) The "&" operators are those originally defined for Ada/SQL. "and" may be used instead, and may be useful in avoiding problems with operator precedence. If one were to (incorrectly) write:

```
SELEC ( NAME & SALARY + COMMISSION . . .,
```

then the "&" operator would be effectively applied before the "+" operator. This would be an error, of course, because no appropriate "+" operator would be defined. If it is desired to use the "&" operator for the above fragment, then parentheses may be used to correctly rewrite it as:

```
SELEC ( NAME & ( SALARY + COMMISSION ) . . .
```

The precedence of the "and" operator is such that it may be used without requiring

parenthesization:

```
SELEC ( NAME and SALARY + COMMISION . . .
```

Release 1 implementations do not recognize "and"s used to separate <value expression>s in a <select list>.

7) ANSI SQL SR8, corresponding to Ada/SQL SR11, includes mention of precision and scale. A concept equivalent to precision is included in the description of an Ada/SQL floating point data type, and scale is not relevant to Ada/SQL data types. (It may become so at a later date, if support for fixed point types is added.)

8) Release 1 implementations do not support null values. Hence, all columns may be considered to be constrained to contain only nonnull values.

9) ANSI SQL GR3, which, like Ada/SQL GR3, applies when R is a grouped table that has zero groups, describes circumstances under which the result of the <query specification> shall contain one row. It does not, however, exhaustively cover all circumstances that are possible for a grouped table with zero groups. Hence, there are conditions left unspecified by ANSI SQL GR3. Furthermore, the values to be contained in that single row, when applicable, are not well-defined in ANSI SQL GR3. Finally, the whole idea of returning a single row for a <query specification> on a table with zero groups is counter-intuitive. Hence, Ada/SQL GR3 specifies that an empty table is to be returned in all cases for a <query specification> on a grouped table with zero groups. Although this differs from ANSI SQL GR3, it corrects the latter's errors of incompleteness, imprecise definition, and departure from expected action. The Ada/SQL interpretation is actually the same as is currently being considered by the ANSI committee for the next version of SQL.

5.26 <table name with optional column list>

Function

Specify a table and, optionally, a list of columns in the table.

Format

<table name with optional column list> ::=
  <table name>
 | [<authorization identifier> -] <table identifier> (<column list>)

<column list> ::=
  <column name> [{ & <column name> } ...]

Effective Ada Declarations

For a table t with authorization identifier a:

```
type TABLE_IDENTIFIER_WITH_COLUMN_LIST_a is private;

type TABLE_NAME_WITH_COLUMN_LIST is private;

type COLUMN_LIST_t is private;

function "-"
         ( LEFT  : AUTHORIZATION_IDENTIFIER_a;
           RIGHT : TABLE_IDENTIFIER_WITH_COLUMN_LIST_a )
 return TABLE_NAME_WITH_COLUMN_LIST;

function t ( LIST : COLUMN_LIST_t )
 return TABLE_IDENTIFIER_WITH_COLUMN_LIST_a;

function t ( LIST : COLUMN_NAME_t )
 return TABLE_IDENTIFIER_WITH_COLUMN_LIST_a;

function t ( LIST : COLUMN_LIST_t ) return TABLE_NAME_WITH_COLUMN_LIST;

function t ( LIST : COLUMN_NAME_t ) return TABLE_NAME_WITH_COLUMN_LIST;

function "&" ( LEFT , RIGHT : COLUMN_NAME_t ) return COLUMN_LIST_t;

function "&"
         ( LEFT  : COLUMN_LIST_t;
           RIGHT : COLUMN_NAME_t ) return COLUMN_LIST_t;
```

Example

```
NEW_EMPLOYEE,
HIS_MANAGER : EMPLOYEE_NAME;
```

```
HIS_SALARY   : EMPLOYEE_SALARY;
 . . .
INSERT_INTO ( EMPLOYEE ( NAME & SALARY & MANAGER ) ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
 . . .
INSERT_INTO ( EXAMPLE-EMPLOYEE ( NAME & SALARY & MANAGER ) ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
 . . .
INSERT_INTO ( EMPLOYEE ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
 . . .
INSERT_INTO ( EXAMPLE.EMPLOYEE ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
```

Syntax Rules

1) The <table name> represented in a <table name with optional column list> is determined as follows:

   Case:

   a) If a <table name> is specified, then that <table name> is the one represented in the <table name with optional column list>.

   b) If a <table name> is not specified, then:

      Case:

      i) If an <authorization identifier> is specified, then the <table name> formed as <authorization identifier>.<table identifier>, using the contained <authorization identifier> and <table identifier>, is the one represented in the <table name with optional column list>.

      ii) If an <authorization identifier> is not specified, then the <table name> formed as the contained <table identifier> is the one represented in the <table name with optional column list>.

2) The same <column name> shall not be specified more than once in the <column list>.

General Rules

None.

Notes

1) <table name with optional column list> conforms to the following parts of ANSI SQL syntax:

   <table name> [(<view column list>)] in <view definition> (6.5)

<table name> [(<insert column list>)] in <insert statement> (8.7)

It has been factored out for Ada/SQL because the effective Ada declarations for both uses are the same.

2) Where both an <authorization identifier> and a <column list> are specified, the effective Ada/SQL <table name> syntax is <authorization identifier>-<table identifier>, rather than <authorization identifier>. <table identifier>. This is required in order to make effective Ada declarations readily possible.

3) Release 1 implementations do not support <authorization identifier>s within <table name with optional column list>s.

6. Schema definition language

6.1 <schema>

Function

Define a <schema>.

Format

<schema> ::=
    <authorization package>
    <schema package> ...

<schema package> ::=
    <schema package declaration>
  [<schema package body>]

Effective Ada Declarations

None.

Example

```
-- <authorization package>:

with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package EXAMPLE_AUTHORIZATION is

  function EXAMPLE is new AUTHORIZATION_IDENTIFIER;

end EXAMPLE_AUTHORIZATION;

-- <schema package> containing only a <schema package declaration>:

package EXAMPLE_TYPES is

  package ADA_SQL is

    type EMPLOYEE_NAME is new STRING ( 1 .. 30 );

    type BOSS_NAME is new EMPLOYEE_NAME;

    type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

    type HOURLY_WAGE_FOR_COMPUTATIONS is new EMPLOYEE_SALARY;
```

```
    subtype HOURLY_WAGE is HOURLY_WAGE_FOR_COMPUTATIONS range 0.00 .. 48.08;

  end ADA_SQL;

end EXAMPLE_TYPES;

-- <schema package> containing a <schema package declaration> (below) and a
--  <schema package body) (follows):

with SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION, EXAMPLE_TYPES;
 use SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION;

package EXAMPLE_SDL is

  package ADA_SQL is

    use EXAMPLE_TYPES.ADA_SQL;

    SCHEMA_AUTHORIZATION : IDENTIFIER := EXAMPLE;

    subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;

    type EMPLOYEE is
      record
        NAME    : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY  : EMPLOYEE_SALARY;
        MANAGER : EMPLOYEE_NAME;
      end record;

    type NEW_EMPLOYEE_FILE is
      record
        NAME    : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY  : EMPLOYEE_SALARY;
        MANAGER : EMPLOYEE_NAME;
      end record;

    type ONE_EMPLOYEE_TABLE is
      record
        NAME    : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY  : EMPLOYEE_SALARY;
        MANAGER : EMPLOYEE_NAME;
      end record;

    type MANAGERS is
      record
        NAME : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
      end record;

  end ADA_SQL;

end EXAMPLE_SDL;
```

```
-- <schema package body>:

with EXAMPLE_SDL_ADA_SQL;
 use EXAMPLE_SDL_ADA_SQL;

package body EXAMPLE_SDL is

begin

  CREATE_VIEW ( MANAGERS ( NAME ),
  AS => SELECT_DISTINCT ( MANAGER,
        FROM  => EMPLOYEE,
        WHERE => IS_NOT_NULL ( MANAGER ) ) );

end EXAMPLE_SDL;
```

Syntax Rules

1) The <authorization package>, <schema package declaration>s, and <schema package body>s within a <schema> need not be part of the same compilation.

2) A <schema package> contained in a <schema> shall contain a <schema authorization clause> naming the same <authorization identifier> as is contained in the <authorization package> contained in the <schema>.

3) A <schema package body> shall contain the same <package identifier> as does the <schema package declaration> contained in the same <schema package>.

General Rules

1) The name of a <schema package> is that of its contained <schema package declaration>.

Notes

1) An Ada/SQL <schema> performs the same functions as does an ANSI SQL <schema>. The major differences between the two <schema>s are:

   a) An ANSI SQL <schema> must be represented in one textual unit; an Ada/SQL <schema> may be divided into separate compilation units in accordance with Ada modularity concepts.

   b) It is necessary to separate an Ada/SQL <schema> into two parts, due to Ada syntax requirements, with some definitions going into the <schema package declaration> and some definitions going into the <schema package body>.

6.1.1 <authorization package>

Function

Declare an <authorization identifier>.

Format

<authorization package> ::=
  with SCHEMA_DEFINITION;
  use SCHEMA_DEFINITION;
  package <package identifier> is
   function <authorization identifier> is new AUTHORIZATION_IDENTIFIER;
  end [<package identifier>] ;

Effective Ada Declarations

Within package SCHEMA_DEFINITION:

```
generic
function AUTHORIZATION_IDENTIFIER return IDENTIFIER;

see also section 5.5 for definition of type IDENTIFIER for representing
<authorization identifier>s
```

Example

```
with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package EXAMPLE_AUTHORIZATION is

  function EXAMPLE is new AUTHORIZATION_IDENTIFIER;

end EXAMPLE_AUTHORIZATION; -- variation: end;
```

Syntax Rules

1) If an <authorization package> contains two <package identifier>s, then both <package identifier>s shall be identical.

2) The <authorization identifier> shall be distinct from the <authorization identifier> declared by any other <authorization package> in the same environment. The concept of environment is implementor-defined.

General Rules

1) The first <package identifier> is declared to be the name of the <authorization package>.

2) An <authorization package> declares its contained <authorization identifier>.

Notes

1) An Ada/SQL <authorization package> performs part of the function of an ANSI SQL <schema authorization clause> (Ada/SQL also contains a <schema authorization clause> to perform the rest of the function). Ada/SQL SR2 corresponds to ANSI SQL SR1 in section 6.1.

2) Release 1 implementations require that each <authorization package> be contained in a separate source file, and that the name of the file be the same as the name of the <authorization package>, possibly augmented with an implementation-dependent indication that the file contains Ada source code. This is done so that the text of an <authorization package> can be found when its name is encountered in a <with clause>.

6.1.2 <schema package declaration>

Function

Declare data types, subtypes, tables, and columns.

Format

<schema package declaration> ::=
  [<context clause>]
    <schema package specification> ;

<schema package specification> ::=
    package <package identifier> is
     [<use clause> ...]
     package ADA_SQL is
      [<use clause> ...]
      [<schema authorization clause>]
       <schema specification element> ...
     end ADA_SQL;
    end [<package identifier>]

<schema authorization clause> ::=
    SCHEMA_AUTHORIZATION : IDENTIFIER := <schema authorization identifier>

<schema authorization identifier> ::=
    <authorization identifier>

<schema specification element> ::=
    <type declaration>
  | <subtype declaration>
  | <number declaration>
  | <table definition>

Effective Ada Declarations

See 5.5 for the effective declaration of type IDENTIFIER within package SCHEMA_DEFINITION.

Example

-- <schema package specification> not containing a <schema authorization -- clause>

```
package EXAMPLE_TYPES is

  package ADA_SQL is

    MAX_NAME_LENGTH : constant := 30;

    type EMPLOYEE_NAME is new STRING ( 1 .. MAX_NAME_LENGTH );

    type BOSS_NAME is new EMPLOYEE_NAME;
```

```
    type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

    type HOURLY_WAGE_FOR_COMPUTATIONS is new EMPLOYEE_SALARY;

    subtype HOURLY_WAGE is HOURLY_WAGE_FOR_COMPUTATIONS range 0.00 .. 48.08;

  end ADA_SQL;

end EXAMPLE_TYPES;

-- <schema package specification> containing a <schema authorization clause>

with SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION, EXAMPLE_TYPES;
 use SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION;

package EXAMPLE_SDL is

  -- variation: use EXAMPLE_TYPES.ADA_SQL;

  package ADA_SQL is

    use EXAMPLE_TYPES.ADA_SQL;

    SCHEMA_AUTHORIZATION : IDENTIFIER := EXAMPLE;

    subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;

    type EMPLOYEE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type NEW_EMPLOYEE_FILE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type ONE_EMPLOYEE_TABLE is
      record
        NAME     : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
        SALARY   : EMPLOYEE_SALARY;
        MANAGER  : EMPLOYEE_NAME;
      end record;

    type MANAGERS is
      record
        NAME : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
      end record;
```

```
  end ADA_SQL;

end EXAMPLE_SDL;
```

Syntax Rules

1) If a <schema package specification> contains two <package identifier>s, then both <package identifier>s shall be identical.

2) Case:

   a) If a <schema package declaration> contains a <schema authorization clause>, then:

      i) It shall be contained within a <schema>.

      ii) It shall contain SCHEMA_DEFINITION as a <package name> and also as a <library package name> not contained in a <package name>.

      iii) It shall contain, as a <package name> and also as a <library package name> not contained in a <package name>, the name of the <authorization package> contained in the same <schema>.

   b) If a <schema package declaration> does not contain a <schema authorization clause>, then:

      i) It is not contained within a <schema>, but may define data types and subtypes for use in other SDL components.

      ii) It shall not contain a <table definition>.

General Rules

1) The first <package identifier> is declared to be the name of the <schema package specification> and of the <schema package declaration>.

Notes

1) The functions of an ANSI SQL <schema element> are encompassed by two Ada/SQL syntactic elements: <schema specification element> and <schema body element>. In addition, <schema specification element> also provides for declaring data types, subtypes, and <named number>s.

2) Release 1 implementations require that each <schema package declaration> be contained in a separate source file, and that the name of the file be the same as the name of the <schema package declaration>, possibly augmented with an implementation-dependent indication that the file contains Ada source code. This is done so that the text of a <schema package declaration> can be found when its name is encountered in a <with clause>.

3) Release 1 implementations do not support <number declaration>s.

6.1.3 <schema package body>

Function

Define uniqueness constraints, viewed tables, and privileges.

Format

```
<schema package body> ::=
    <context clause>
    package body <package identifier> is
     [ <use clause> ... ]
     [ <number declaration> ... ]
    begin
     <schema body element> ...
    end [ <package identifier> ] ;

<schema body element> ::=
    <view definition>
  | <privilege definition>
  | <unique constraint definition>
```

Effective Ada Declarations

None.

Example

```
with EXAMPLE_SDL_ADA_SQL;
 use EXAMPLE_SDL_ADA_SQL;

package body EXAMPLE_SDL is

  -- variations: use EXAMPLE_SDL_ADA_SQL; -- <use clause>
                 ONE : constant := 1; -- <number declaration>

begin

  CREATE_VIEW ( MANAGERS ( NAME ), -- <view definition>
  AS => SELECT_DISTINCT ( MANAGER,
        FROM  => EMPLOYEE,
        WHERE => IS_NOT_NULL ( MANAGER ) ) );

  GRANT ( SELEC , ON => MANAGERS , TO => PUBLIC ); -- <privilege definition>

  CONSTRAINTS ( NEW_EMPLOYEE_FILE , UNIQUE ( SALARY & MANAGER ) );
    -- <unique constraint definition>

end EXAMPLE_SDL; -- variation: end;
```

Syntax Rules

1) If a <schema package body> contains two <package identifier>s, then both <package identifier>s shall be identical.

2) Let P be the <package identifier>. A <schema package body> shall contain P_ADA_SQL as a <package name> and also as a <library package name> not contained in a <package name>. P_ADA_SQL is called the *Ada/SQL definition package* associated with the <schema package body>.

General Rules

None.

Notes

1) In a runtime system, the P_ADA_SQL package referred to in SR2 defines all the database names and operations used in the <schema body element>s of a <schema package body>. In a preprocessed system, references to P_ADA_SQL can be deleted by the preprocessor.

2) A <context clause> is required in a <schema package body>, even though it is optional in a <schema package declaration>. This is because P_ADA_SQL (as defined in SR2) must be named in the <context clause> of a <schema package body>.

3) Release 1 implementations do not support <schema package body>s – all information from a <schema> must be replicated in the database structure in a way that is not defined by those implementations.

6.1.4 <context clause>

Function

Identify <schema package>s and predefined environment packages that are referenced from an <Ada/SQL compilation unit>.

Format

<context clause> ::=
{ <with clause> [<use clause> ...] } ...

<with clause> ::=
with <unit simple name> [{ , <unit simple name> } ...] ;

<unit simple name> ::=
<library package name>
| <non Ada/SQL library unit name>

<use clause> ::=
use <package name> [{ , <package name> } ...] ;

Effective Ada Declarations

None.

Example

– <context clause> with no <use clause>:

```
with SCHEMA_DEFINITION;

-- <context clause> with duplications, several <with clause>s and
--  <use clause>s:

with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;
with EXAMPLE_AUTHORIZATION, EXAMPLE_TYPES;
with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION, EXAMPLE_AUTHORIZATION;
 use EXAMPLE_TYPES;

-- <use clause> not contained in a <context clause>:

use EXAMPLE_TYPES.ADA_SQL;
```

Syntax Rules

1) Case:

a) If a <context clause> is contained in a <schema package declaration>, then each <unit simple name> contained within it shall be a <library package name> which is the name of an <authorization package>, a <schema package>, or a package from the predefined Ada/SQL environment.

b) If a <context clause> is contained in a <schema package body>, then each <unit simple name> contained within it shall be a <library package name> which is the name of an <authorization package>, a <schema package>, the Ada/SQL definition package associated with the <schema package body>, or a package from the predefined Ada/SQL environment.

c) If a <context clause> is contained in a <global variable package>, then:

Case:

i) If the <global variable package> contains any <correlation name declaration>s, then each <unit simple name> contained within the <context clause> shall be a <library package name> which is the name of a <schema package>, the Ada/SQL definition package associated with the <global variable package>, or a package from the predefined Ada/SQL environment.

ii) If the <global variable package> does not contain any <correlation name declaration>s, then each <unit simple name> contained within the <context clause> shall be a <library package name> which is the name of a <schema package> or a package from the predefined Ada/SQL environment.

d) If a <context clause> is contained in an <Ada/SQL DML unit>, then each <unit simple name> contained within it shall be a <library package name> which is the name of an <authorization package>, a <schema package>, a <global variable package>, the Ada/SQL definition package associated with the <Ada/SQL DML unit>, or a package from the predefined Ada/SQL environment; or it shall be a <non Ada/SQL library unit name> which is the name of an Ada library_unit such that at least one of the following conditions is true:

i) The library_unit is not a package_declaration.

ii) If (i) is false, then the visible part of the package_declaration does not contain any basic_declarative_items that are not use_clauses.

iii) If (i)-(ii) are false, then all of the following conditions are true:

1) The tokens comprising the first basic_declarative_item are not: **function** identifier **is new** AUTHORIZATION_IDENTIFIER ;

2) The tokens comprising the the first basic_declarative_item this is not a use_clause are not: **package** ADA_SQL **is**

3) The first basic_declarative_item that is not a use_clause is not an object_declaration

2) If a <use clause> is contained in a <context clause>, then each <package name> contained within it shall be of the form <unit simple name>.

3) If a <use clause> is not contained in a <context clause>, then each <unit simple name> within it shall be of the form <library package name>.

4) Each <unit simple name> contained in a <use clause> shall be the same as a <unit simple name> contained in a textually prior <with clause> contained in the same <schema package>, <global variable package>, or <Ada/SQL DML unit>, as appropriate.

5) Case:

   a) If a <with clause> or a <use clause> is contained in a <context clause> of a <schema package declaration>, then it *applies* to both the <schema package declaration> and also to the <schema package body>, if any, contained in the same <schema package>.

   b) If a <with clause> or a <use clause> is contained in a <context clause> of a <schema package body>, <global variable package>, or an <Ada/SQL DML unit>, then it *applies* to that <schema package body>, <global variable package>, or <Ada/SQL DML unit>.

   c) If a <use clause> is contained in a <local variable package>, then it *applies* to the <local variable package> in which it is contained.

   d) If a <use clause> is not contained in a <context clause> or a <local variable package>, then it *applies* to the <schema package declaration>, <schema package body>, <global variable package>, or <Ada/SQL DML unit> in which it is contained.

6) A *library package* is an <authorization package>; a <schema package>; a <global variable package>; the Ada/SQL definition package associated with a <schema package body>, a <global variable package>, or an <Ada/SQL DML unit>; or a package from the predefined Ada/SQL environment. The appearance of a <library package name> within a <with clause> defines that <library package name> as denoting the library package with the same name. This definition is valid from the point of definition to the end of the <schema package declaration>, <schema package body>, <global variable package>, or <Ada/SQL DML unit> to which the <with clause> applies.

7) An *innermost package* is an <authorization package>; a <global variable package>; a <local variable package>; the Ada/SQL definition package associated with a <schema package body>, a <global variable package>, or an <Ada/SQL DML unit>; a package from the predefined Ada/SQL environment; or the nested ADA_SQL package in a <schema package declaration>. The appearance of a <package name> within a <use clause> defines that <package name> as denoting an innermost package if:

a) The <package name> is of the form <unit simple name>, and the <unit simple name> is the name of an <authorization package>; a <global variable package>; the Ada/SQL definition package associated with the containing <schema package body>, <global variable package>, or <Ada/SQL DML unit>; or a package from the predefined Ada/SQL environment (the innermost package denoted is the one named), or

b) The <package name> is of the form <unit simple name>.ADA_SQL, and the <unit simple name> is the name of a <schema package> (the innermost package denoted is the nested ADA_SQL package of the <schema package declaration>), or

c) The <package name> is of the form ADA_SQL and one of the following is true:

   i) The <package name> is contained in an <Ada/SQL DML unit> that also contains a preceding <local variable package> (the innermost package denoted is that <local variable package>), or

   ii) The <package name> is contained in a <schema package body> (the innermost package denoted is the nested ADA_SQL package in the <schema package declaration> contained in the same <schema package>).

This definition is valid from the point of definition to the end of the <schema package declaration>, <schema package body>, <global variable package>, <local variable package>, or <Ada/SQL DML unit> to which the <use clause> applies.

8) The effect of <with clause>s and <use clause>s for <schema package>s and <global variable package>s are in accordance with Ada rules. Simplifications are made for <Ada/SQL DML unit>s, however: (1) <with clause>s and <use clause>s do not apply to subunits, (2) <non Ada/SQL package name>s contained within a <use clause> do not affect Ada/SQL syntax, and (3) Ada use_clauses that are not Ada/SQL <use clause>s do not affect Ada/SQL syntax. The visibility of <identifier>s contained within an <Ada/SQL DML unit> shall be the same under the simplified Ada/SQL rules as under complete Ada rules.

General Rules

None.

Notes

1) <schema package>s and <global variable package>s, because their use is limited to Ada/SQL constructs, may reference only other Ada/SQL packages. <Ada/SQL DML unit>s, however, may reference arbitrary Ada library units. SR1d is designed to enable Ada/SQL automated tools to readily determine whether or not a unit referenced is an Ada/SQL package.

2) The syntax rules of <use clause>, as well as of <package name>, involve several simplifications over Ada syntax. A <unit simple name> in a <use clause> is not permitted to be the name of the containing program unit. Also, ADA_SQL as a <package name> in a <use clause> is only permitted where it refers to a <local variable package> contained in the same <Ada/SQL DML unit> or to the nested ADA_SQL package contained in the same <schema package>. Since so many things

are named ADA_SQL, this restriction is designed to minimize confusion about what is being referenced, as well as to simplify the development of Ada/SQL automated tools.

3) A non Ada/SQL library unit P could have a package named ADA_SQL declared within it. P is a valid <unit simple name> (a <non Ada/SQL library unit name>), so in the absence of SR3 P.ADA_SQL in a <use clause> is syntactically ambiguous: it could be taken either as <unit simple name>. ADA_SQL or as a <non Ada/SQL package name>. SR3 forbids a <unit simple name> in this context from being a <non Ada/SQL library unit name>, so it restricts the syntactic interpretation to the desired one: P.ADA_SQL is a <non Ada/SQL package name>.

4) The rules expressed in SR5 for how <with clause>s and <use clause>s contained within a <context clause> apply to units are a simplification of Ada rules, in that subunits are not considered by Ada/SQL. The other simplifications mentioned in SR8 are consequences of other syntax. For example, <use clause>s can only appear at the beginning of compilation units. These simplifications are designed to enable Ada/SQL automated tools to be written without requiring them to understand all of Ada library unit rules and block structure.

5) Release 1 implementations only process the first <with clause> and the first <use clause> (and then only if it immediately follows the first <with clause>) of a <context clause> contained in an <Ada/SQL DML unit>. It is assumed that units named in later <with clause>s do not apply to Ada/SQL – these units would satisfy one of conditions (i) - (iii) in SR1d. Release 1 implementations therefore do not have to check the conditions. When processing <Ada/SQL DML unit>s, Release 1 implementations also do not process <use clause>s not contained in <context clause>s. Instead, they consider that if <library package name> P appears in the <use clause> processed, and P contains a nested ADA_SQL package, then P.ADA_SQL also appears in a <use clause> that applies to the <Ada/SQL DML unit>.

6.1.5 <type declaration>

Function

Declare a data type.

Format

<type declaration> ::=
    <full type declaration>

<full type declaration> ::=
    type <type identifier> is <type definition> ;

<type identifier> ::=
    <program identifier>

<type definition> ::=
    <data type>
  | <derived type definition>

<derived type definition> ::=
    new <subtype indication>

Effective Ada Declarations

In the DATABASE predefined package:

```
type INT is range implementation_defined;

subtype SMALLINT is INT range implementation_defined;

type DOUBLE_PRECISION is digits implementation_defined
 range implementation_defined;

subtype REAL is DOUBLE_PRECISION digits implementation_defined
 range implementation_defined;

-- The ranges and accuracies (for floating point) of these types/subtypes
-- are defined such that they represent the maximums that may be used with
-- the correspondingly-named SQL data types of the underlying database
-- implementation.  If the ranges and accuracies supported by the Ada
-- system are at least as great as those supported by the database, then
-- the Ada types/subtypes will match the database data types.  If the Ada
-- system is more restrictive, then the definitions of these types/-
-- subtypes will (obviously) reflect those restrictions.
```

The following definitions from STANDARD may also be used with Ada/SQL:

BOOLEAN, INTEGER, FLOAT, CHARACTER, NATURAL, POSITIVE, STRING

Example

```
type EMPLOYEE_SALARY is digits 7 range 0.00 .. 99_999.99;

type HOURLY_WAGE_FOR_COMPUTATIONS is new EMPLOYEE_SALARY;
```

Syntax Rules

1) The <type identifier> shall not end in the characters _NOT_NULL or _NOT_NULL_UNIQUE.

2) The <type identifier> shall not be identical to the name of any data type, subtype, table, or <named number> declared by any other <schema specification element> in the containing <schema package specification>.

General Rules

1) A <type declaration> declares a data type. The <type identifier> is declared to be the name of the data type.

   Case:

   a) A <type declaration> containing an <unconstrained character string definition> declares a character string type with component subtype and index subtype given by the <character string type> containing the <unconstrained character string definition>. The <type identifier> denotes the character string type.

   b) A <type declaration> containing a <constrained character string definition> declares both a character string type and a subtype. The character string type is an implicitly declared anonymous type; this type is defined by an (implicit) <unconstrained character string definition>, in which the <component subtype indication> is that of the <constrained character string definition>, and in which the (anonymous) <type mark> of the <index subtype definition> denotes the subtype defined by the <index constraint>. The character string subtype declared by the <type declaration> is the subtype obtained by imposition of the <index constraint> (as in a <subtype indication>) on the implicitly declared character string type. The <type identifier> denotes the character string subtype.

   c) A <type declaration> containing an <integer type> declares both an integer type and a subtype. The integer type is an implicitly declared anonymous type, with representation selected by the implementation to include at least the values specified by the <range constraint>. The integer subtype declared by the <type declaration> is the subtype obtained by imposition of the <range constraint> (as in a <subtype indication>) on the implicitly declared integer type. The <type identifier> denotes the integer subtype.

   d) A <type declaration> containing a <floating point type> declares both a floating point type and a subtype. The floating point type is an implicitly declared anonymous type, with representation selected by the implementation to provide at least the accuracy specified by the <floating point constraint>, and at least the range of numbers that would be required if the <floating point constraint> did not contain a <range constraint> (see 5.5.3). The floating point subtype declared by the <type declaration> is the subtype obtained by imposition of the <floating point constraint> (as in a <subtype indication>) on the implicitly declared floating

point type. The <type identifier> denotes the floating point subtype.

e) A <type declaration> containing an <enumeration type> declares an enumeration type, with values given by the <enumeration literal specification>s contained within the <enumeration type>. The <type identifier> denotes the enumeration type.

f) A <type declaration> containing a <derived type definition> declares both a type and a subtype. The type declared is an implicitly declared anonymous type, belonging to the same class of types (character string, integer, floating point, or enumeration) as does the data type denoted by the <type mark> of the <subtype indication>, with the same set of possible values as that data type. The subtype declared is the result of imposing any <constraint>s denoted by the <type mark> and/or included in the <subtype indication> on the implicitly declared type. The <type identifier> denotes the subtype.

Notes

None.

6.1.6 <subtype declaration>

Function

Declare a subtype.

Format

<subtype declaration> ::=
subtype <type identifier> **is** <subtype indication> ;

Effective Ada Declarations

See subtypes SMALLINT and REAL declared in 6.1.5

Example

```
subtype HOURLY_WAGE is HOURLY_WAGE_FOR_COMPUTATIONS range 0.00 .. 48.08;

subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;
```

Syntax Rules

1) The <type identifier> shall not be identical to the name of any data type, subtype, table, or <named number> declared by any other <schema specification element> in the containing <schema package specification>.

2) Let I denote the <type identifier> following the reserved word **subtype**, and let M denote the <type identifier> of the <type mark> contained within the <subtype indication>.

   Case:

   a) If the last characters of I are neither _NOT_NULL nor _NOT_NULL_UNIQUE, then the last characters of M shall be neither _NOT_NULL nor _NOT_NULL_UNIQUE.

   b) If I is of the form V_NOT_NULL, then:

      i) M shall be identical to V.

      ii) The <subtype indication> shall not contain a <constraint>.

   c) If I is of the form V_NOT_NULL_UNIQUE, then:

      i) M shall either be identical to V or shall be V_NOT_NULL.

      ii) The <subtype indication> shall not contain a <constraint>.

General Rules

1) A <subtype declaration> declares the subtype defined by its contained <subtype indication>. The <type identifier> is declared to be the name of the subtype.

Notes

1) The _NOT_NULL and _NOT_NULL_UNIQUE suffixes are used to declare database columns with the similarly-named constraints. SR2 establishes a hierarchy for subtype names that is consistent with the semantics of the corresponding constraints: _NOT_NULL is a stronger constraint than is no constraint, and _NOT_NULL_UNIQUE is a stronger constraint than is _NOT_NULL.

6.1.7 <number declaration>

Function

Declare a <named number>.

Format

<number declaration> ::=
    <named number list> : constant := <value specification> ;

<named number list> ::=
    <named number> [{ , <named number> } ...]

<named number> ::=
    <program identifier>

Effective Ada Declarations

See 5.5.1, 5.5.2, and 5.5.3 for <named number>s that are declared to reflect system limits.

Example

```
PI             : constant := 3.14159_26536; -- a floating point number
TWO_PI         : constant := 2.0 * PI;      -- a floating point number
MAX            : constant := 500;           -- an integer number
ONE, UN, EINS  : constant := 1;             -- three different names for 1
```

Syntax Rules

1) Each <named number> shall be distinct from the name of any other data type, subtype, table, or <named number> declared in the containing <schema package specification>.

2) The <value specification> shall not contain a <value specification primary> other than a <literal> or a <variable specification> containing a <named number> but not containing an <indicator specification>.

3) If a <named number> is declared by a <number declaration> containing a <floating point literal> or a floating point <named number>, then it is a floating point <named number>; otherwise, it is an integer <named number>.

4) Arithmetic on integer <named number>s (and <integer literal>s) is performed without regard to the constraints of any particular data type, as if they were of a *universal integer* data type. When interpretation as a value of a specific integer data type is required by the context, an integer <named number> or an otherwise untyped expression of integer <named number>s and/or <integer literal>s is taken to be of that data type.

5) Arithmetic on floating point <named number>s (and <floating point literal>s) is performed without regard to the constraints of any particular data type, as if they were of a *universal floating point* data type. When interpretation as a value of a specific floating point data type is required by the context, a floating point <named number> or an otherwise untyped expression of floating point <named number>s and/or <floating point literal>s is taken to be of that data type. (Universal floating point expressions may also contain integer <named number>s and <integer literal>s in certain contexts; see 5.6, "<value specification>", and 5.9, "<value expression>".)

General Rules

1) A <number declaration> declares its contained <named number>s. The value of the <named number>s is that of the <value specification>.

2) The value of an integer <named number> or an untyped expression of integer <named number>s and/or <integer literal>s, when taken to be of a specific integer data type, shall be exact. The value shall belong to the data type; otherwise, the CONSTRAINT_ERROR exception is raised.

3) The value of a floating point <named number> or an untyped expression containing floating point <named number>s and/or <floating point literal>s, when taken to be of a specific floating point data type, shall be within the accuracy of that data type. The value shall belong to the data type; otherwise, the CONSTRAINT_ERROR exception is raised.

Notes

1) Release 1 implementations do not support <number declaration>s.

2) Ada/SQL SR2 ensures that the <value specification> defining the value of a <named number> is an Ada static expression.

3) Ada/SQL SRs 3-5 express aspects of Ada/SQL's strong typing, consistent with Ada's implicit type conversions from types *universal_integer* and *universal_real*.

4) Raising CONSTRAINT_ERROR in GR2 and GR3 corresponds to the Ada semantics on implicit type conversions.

6.2 <table definition>

Function

Define a base table or a viewed table.

Format

<table definition> ::=
  type <table name> is
   record
    <table element> ...
   end record;

<table element> ::=
  <column definition> ;

Effective Ada Declarations

None.

Example

```
type EMPLOYEE is
  record
    NAME    : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
    SALARY  : EMPLOYEE_SALARY;
    MANAGER : EMPLOYEE_NAME;
  end record;
```

Syntax Rules

1) The <table name> shall not contain an <authorization identifier>.

2) The <table name> shall be different from the <table name> of any other <table definition> in the containing <schema>.

3) The <table name> shall not be identical to the name of any data type, subtype, table, or <named number> declared by any other <schema specification element> in the containing <schema package specification>.

4) The description of the table defined by a <table definition> includes the name <table name> and the column description specified by each <column definition>. The i-th column description is given by the i-th <column definition>.

General Rules

1) A <table definition> defines either a base table or a viewed table. If the containing <schema package> contains a <schema package body>, and then that <schema package body> contains a <view definition> containing a <table identifier> in the <table name with optional column list>, that is the same <table identifier> as that contained in the <table name> of the <table definition>, then the <table definition> defines a viewed table. Otherwise, the <table definition> defines a base table.

Notes

1) The Ada/SQL <table definition> conforms to the ANSI SQL <table definition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	2
SR2	SR2	3
—	SR3	4
SR3	SR4	5
GR1	GR1	6

2) ANSI SQL permits an <authorization identifier> in the <table name>, but it is restricted to be the same as <schema authorization identifier> of the containing <schema>. Ada syntax provides no way to conveniently include an <authorization identifier> in the <table name> contained in a <table definition>, and no capability is lost by not allowing it, since it is restricted to only one value anyway.

3) ANSI SQL SR2 also requires that the <table name> be different from that of any <view definition> in the containing <schema>. As discussed in Note 6, below, Ada/SQL requires that <table definition>s also be given for views, not just for base tables. Consequently, Ada/SQL SR2 is sufficient to prevent duplication of table and view names.

4) Ada/SQL <schema>s define other entities besides tables and views, and Ada syntax requires that all names be unique within the same <schema package specification>.

5) The ANSI SQL format for <table element> includes <unique constraint definition>. Ada syntax does not permit <unique constraint definition>s to be conveniently included within a <table definition>, so <unique constraint definition>s are instead placed within <schema package body>s.

6) In ANSI SQL, a <table definition> defines a base table, while a <view definition> defines a viewed table. In Ada/SQL, base tables and viewed tables each require a <table definition>. The presence or absence of a corresponding <view definition> determines whether a table is a viewed table or a base table.

   The reasons for requiring a <table definition> for views are analogous to the reasons that Ada

separates entities into specifications and bodies. The <table definition> for a view defines the names and data types of its columns, and so can be considered as the specification of the view. A <view definition> provides the detailed instructions on how to materialize the data in the view, and so can be considered as the body of the view. This dichotomy is carried further by the Ada/SQL syntax: <table definition>s are placed in <schema package specification>s, while <view definition>s are placed in <schema package body>s. If the way in which data is materialized for a view changes, without affecting the specification of the view, then only the <view definition> in a <schema package body> is changed; the <table definition> in a <schema package specification> is unaffected. The semantics of Ada recompilation rules for the <schema package> are then also the desired semantics for programs using the view: programs using the view do not have to be changed (or even recompiled), since only the body has changed, without changing the specification.

6.3 <column definition>

Function

Define a column of a table.

Format

<column definition> ::=
    <column name> : <subtype indication>

Effective Ada Declarations

None.

Example

```
NAME   : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
SALARY : EMPLOYEE_SALARY;
```

Syntax Rules

1) The <column name> shall be different from the <column name> of any other <column definition> in the containing <table definition>.

2) The i-th column of the table is described by the i-th <column definition> in the <table definition>. The name and subtype of the column are specified by the <column name> and <subtype indication>, respectively.

3) If the <type mark> of the <subtype indication> ends with the characters _NOT_NULL or _NOT_NULL_UNIQUE, then the containing <table definition> shall define a base table and the column is constrained to contain only nonnull values.

4) Let C denote the <column name> of a <column definition> contained in a <table definition> of table T. If the <type mark> of the <subtype indication> ends with the characters _NOT_NULL_UNIQUE, then the containing <table definition> shall define a base table and the following <unique constraint definition> is implicit:

```
CONSTRAINTS ( T , UNIQUE ( C ) );
```

5) The description of the column defined by a <column definition> includes the name <column name> and the subtype specified by the <subtype indication>.

6) If the <type mark> of the <subtype indication> denotes an unconstrained character string subtype, then the <subtype indication> shall also contain an <index constraint> to define the index bounds of the column.

General Rules

1) If a column is constrained to contain only nonnull values, then the constraint is effectively checked after the execution of each <SQL statement>. Any <SQL statement> that would cause the constraint to be violated has no effect on the database, and instead causes the CONSTRAINT_VIOLATION exception to be raised.

Notes

1) The Ada/SQL <column definition> conforms to the ANSI SQL <column definition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	2
SR2	SR2	3
SR3-SR4	SR3-SR4	4
SR5	SR5	3
—	SR6	5
GR1	GR1	-

2) The original specification of Ada/SQL allowed a list of <column name>s in a <column definition>, as in an Ada component_declaration. Only allowing a single <column name> per <column definition>, as is done here, is closer to SQL syntax, however.

3) An ANSI SQL column description includes the data type of the column, taken from a very limited set of predefined data types; Ada/SQL extends this to include the subtype of the column, with user-defined data types permitted.

4) Ada/SQL <table definition>s apply to viewed tables as well as to base tables. ANSI SQL provides no way to specify not null or uniqueness constraints for columns of a view, so we disallow such specifications in the <column definition>s for a viewed table.

5) Ada/SQL SR6 is a consequence of Ada rules that the subtype of a record component must be constrained.

6.4 <unique constraint definition>

Function

Specify a uniqueness constraint for a table.

Format

<unique constraint definition> ::=
CONSTRAINTS (<table name> , UNIQUE (<unique column list>)) ;

<unique column list> ::=
<column name> [{ & <column name> } ...]

Effective Ada Declarations

For a table t:

```
type UNIQUE_COLUMN_LIST_t is private;

function UNIQUE ( COLUMNS : COLUMN_LIST_t ) return UNIQUE_COLUMN_LIST_t;

function UNIQUE ( COLUMN : COLUMN_NAME_t ) return UNIQUE_COLUMN_LIST_t;

procedure CONSTRAINTS
          ( TABLE   : TABLE_NAME_t;
            COLUMNS : UNIQUE_COLUMN_LIST_t );
```

see also section 5.26 for definition of type COLUMN_LIST_t and the "&" operators

Example

```
CONSTRAINTS ( EMPLOYEE , UNIQUE ( NAME ) );

CONSTRAINTS ( EMPLOYEE , UNIQUE ( NAME & MANAGER ) );
```

Syntax Rules

1) If the <table name> contains an <authorization identifier>, then that <authorization identifier> shall be the same as the <schema authorization identifier> of the containing <schema>.

2) The <table identifier> contained in the <table name> shall be the same as the <table identifier> contained in a <table definition> defining a base table within the same <schema package>.

3) Let T denote the table specified by the <table name>.

4) Each <column name> in the <unique column list> shall identify a column of T, and the same column shall not be identified more than once.

5) The <column definition> for each <column name> in the <unique column list> shall indicate that null values are not permitted in the column.

General Rules

1) Let "designated columns" denote the columns identified by the <column name>s of the <unique column list>.

2) T is constrained to contain no rows that are duplicates with respect to the designated columns. Two rows are duplicates if the value of each designated column in the first row is equal to the value of the corresponding column in the second row. The constraint is effectively checked after the execution of each <SQL statement>. Any <SQL statement> that would cause the constraint to be violated has no effect on the database, and instead causes the CONSTRAINT_VIOLATION exception to be raised.

Notes

1) The Ada/SQL <unique constraint definition> conforms to the ANSI SQL <unique constraint definition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1-SR2	2
SR1-SR3	SR3-SR5	-
GR1-GR2	GR1-GR2	-

2) In ANSI SQL, a <unique constraint definition> is contained within the applicable <table definition>. Ada syntax prevents this from being the case in Ada/SQL, where a <unique constraint definition> is contained in the <schema package body> corresponding to the <schema package declaration> containing the applicable <table definition>. Ada/SQL SR1 and SR2 ensure that a <unique constraint definition> is indeed placed in the same <schema package> as its applicable <table definition>. Also, uniqueness constraints may only be specified for a base table, not for a viewed table, as with ANSI SQL.

6.5 <view definition>

Function

Define a viewed table.

Format

<view definition> ::=
CREATE_VIEW (<table name with optional column list> ,
AS => <query specification> [,
WITH_CHECK_OPTION => ENABLED]) ;

Effective Ada Declarations

```
type OPTION_STATE is ( ENABLED , DISABLED );

procedure CREATE_VIEW
          ( TABLE             : TABLE_NAME_WITH_COLUMN_LIST;
            AS                : QUERY_SPECIFICATION;
            WITH_CHECK_OPTION : OPTION_STATE := DISABLED );
```

Example

```
type NAMES_ONLY is
  record
    NAME    : EMPLOYEE_NAME;
    MANAGER : EMPLOYEE_NAME;
  end record;
. . .
CREATE_VIEW ( NAMES_ONLY,
AS => SELEC ( NAME & MANAGER,
      FROM => EMPLOYEE ) );
. . .
type SUPERVISORS is
  record
    EMP        : EMPLOYEE_NAME;
    SUPERVISOR : EMPLOYEE_NAME;
  end record;
. . .
CREATE_VIEW ( SUPERVISORS ( EMP & SUPERVISOR ),
AS => SELEC  ( NAME & MANAGER,
      FROM  => EMPLOYEE,
      WHERE => NE ( NAME , MANAGER ) ),
WITH_CHECK_OPTION => ENABLED );
```

Syntax Rules

1) If the <table name with optional column list> contains an <authorization identifier>, then that <authorization identifier> shall be the same as the <schema authorization identifier> of the

containing <schema>.

2) The <table identifier> contained in the <table name with optional column list> shall be different from the <table identifier> contained in any other <view definition>'s <table name with optional column list> in the containing <schema>.

3) The <table identifier> contained in the <table name with optional column list> shall be the same as the <table identifier> contained in a <table definition> defining a table within the same <schema package>. Let that <table definition> be called the *associated* <table definition>.

4) If the <query specification> is updatable, then the viewed table is an updatable table. Otherwise, it is a read-only table.

5) If any two columns in the table specified by the <query specification> have the same <column name>, or if any column of that table is an unnamed column, then a <column list> shall be specified within the <table name with optional column list>.

6) The number of <column name>s in the <table name with optional column list> shall be the same as the degree of the table specified by the <query specification>.

7) The description of the table defined by a <view definition> is the same as that defined by the associated <table definition>. The data type of the i-th column in the table specified by the <query specification> shall be the same as the data type of the i-th column defined by the associated <table definition>.

   Case:

   a) If a <column list> is contained in the <table name with optional column list>, then the i-th <column name> in that <column list> shall be the same as the i-th <column name> in the associated <table definition>.

   b) If a <column list> is not contained in the <table name with optional column list>, then the <column name> of the i-th column of the <query specification> shall be the same as the i-th <column name> in the associated <table definition>.

8) If the <query specification> contains a <group by clause> or a <having clause> that is not contained in a <subquery>, then the viewed table defined by the <view definition> is a grouped view.

9) If WITH_CHECK_OPTION => ENABLED is specified, then the viewed table shall be updatable.

General Rules

1) A <view definition> defines a viewed table. The viewed table, V, is the table that would result if the <query specification> were executed. Whether a viewed table is materialized is implementor-defined.

2) If V is updatable, then let T denote the table identified by the <table name> specified in the <from clause> in the <query specification>. For each row in V, there is a corresponding row in T from which the row of V is derived. For each column in V, there is a corresponding column in T from which the column of V is derived. The insertion of a row into V is an insertion of a corresponding row into T. The deletion of a row from V is a deletion of the corresponding row in T. The updating of a column of a row in V is an updating of the corresponding row in T.

3) Case:

   a) If WITH_CHECK_OPTION => ENABLED is specified and the <query expression> specifies a <where clause>, then an <insert statement>, an <update statement: positioned>, or an <update statement: searched> to the view shall not result in the creation of a row for which that <where clause> is false. Any such statement that would cause the <where clause> to be false has no effect on the database, and instead causes the CONSTRAINT_VIOLATION exception to be raised.

   b) If WITH_CHECK_OPTION => ENABLED is not specified, then the <view definition> shall not constrain the data values that may be inserted into an updatable viewed table.

NOTE: See General Rule 2 of 8.7, "<insert statement>", General Rule 9 of 8.11 "<update statement: positioned>", and General Rule 4 of 8.12, "<update statement: searched>".

Notes

1) The Ada/SQL <view definition> conforms to the ANSI SQL <view definition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	2
SR2	SR2-SR3	3
SR3-SR4	SR4-SR5	-
SR5	—	4
SR6	SR6	-
SR7	SR7	5
SR8-SR9	SR8-SR9	-
GR1	GR1	-
GR2	GR2	6
GR3	GR3	-

2) Ada/SQL <view definition> syntax contains a <table name with optional column list>, which contains (in effect) the <table name> and <view column list> of the ANSI SQL syntax. This is done to factor out definitions in common with the <insert statement>.

3) ANSI SQL views are defined only by their <view definition>; an Ada/SQL view also has an associated <table definition>. Ada/SQL SR2 ensures that each view has only a single <view definition>; Ada/SQL SR3 establishes the association between a <view definition> and a <table definition>.

4) ANSI SQL SR5 is not required in Ada/SQL; it is covered in 5.26, "<table name with optional column list>".

5) In ANSI SQL, the description of a view comes from its <view definition>. In Ada/SQL, the description is provided by the associated <table definition>. Ada/SQL SR7 ensures that the description that ANSI SQL would infer from a <view definition> is consistent with the description given by the associated <table definition>.

6) The first sentence of ANSI SQL GR2 includes the phrase "in the first <from clause>", while Ada/SQL GR2 omits "first". It is a consequence of other syntax rules that an updatable <query specification> contains exactly one <from clause>.

6.6 <privilege definition>

Function

Define privileges.

Format

<privilege definition> ::=
GRANT (<privileges> ,
ON => <table name> ,
TO => <grantee> [{ & <grantee> } ...] [,
WITH_GRANT_OPTION => ENABLED]) ;

<privileges> ::=
ALL_PRIVILEGES
| <action> [{ & <action> } ...]

<action> ::=
SELEC | INSERT | DELETE
| UPDATE [(<grant column list>)]

<grant column list> ::=
<column name> [{ & <column name> } ...]

<grantee> ::=
PUBLIC | <authorization identifier>

Effective Ada Declarations

```
type AUTHORIZATION_IDENTIFIER_LIST is private;

function PUBLIC return AUTHORIZATION_IDENTIFIER_LIST;

function "&" ( LEFT , RIGHT : AUTHORIZATION_IDENTIFIER_LIST )
 return AUTHORIZATION_IDENTIFIER_LIST;

type ALL_PRIVILEGES_TYPE is ( ALL_PRIVILEGES );

procedure GRANT
          ( PRIVILEGES        : ALL_PRIVILEGES_TYPE;
            ON                : TABLE_NAME;
            TO                : AUTHORIZATION_IDENTIFIER_LIST;
            WITH_GRANT_OPTION : OPTION_STATE := DISABLED );
```

see also section 6.5 for definition of type OPTION_STATE

For a table t:

```
type PRIVILEGES_t is private;
```

```
function SELEC return PRIVILEGES_t;

function INSERT return PRIVILEGES_t;

function DELETE return PRIVILEGES_t;

function UPDATE return PRIVILEGES_t;

function UPDATE ( COLUMNS : COLUMN_LIST_t ) return PRIVILEGES_t;

see also section 5.26 for definition of type COLUMN_LIST_t and the
 "&" operators on <column name>s

function UPDATE ( COLUMN : COLUMN_NAME_t ) return PRIVILEGES_t;

function "&" ( LEFT , RIGHT : PRIVILEGES_t ) return PRIVILEGES_t;

procedure GRANT
          ( PRIVILEGES        : PRIVILEGES_t;
            ON                : TABLE_NAME_t;
            TO                : AUTHORIZATION_IDENTIFIER_LIST;
            WITH_GRANT_OPTION : OPTION_STATE := DISABLED );
```

Example

```
GRANT ( ALL_PRIVILEGES,
  ON => EMPLOYEE,
  TO => MANAGEMENT,
  WITH_GRANT_OPTION => ENABLED );

GRANT ( SELEC,
  ON => EMPLOYEE,
  TO => PUBLIC );

GRANT ( SELEC & UPDATE,
  ON => EMPLOYEE,
  TO => PAYROLL );

GRANT ( SELEC & UPDATE ( NAME ),
  ON => EMPLOYEE,
  TO => SECURITY );

GRANT ( SELEC & INSERT & DELETE & UPDATE ( NAME & MANAGER ),
  ON => EMPLOYEE,
  TO => PERSONNEL );
```

Syntax Rules

1) Let T denote the table identified by the <table name>. The <privileges> sp cify one or more privileges on T.

2) UPDATE (<grant column list>) specifies the UPDATE privilege on each column of T identified in the <grant column list>. Each <column name> in the <grant column list> shall identify a column of T. If the <grant column list> is omitted, then UPDATE specifies the UPDATE privilege on all columns of T.

3) The applicable <privileges> for a reference to a <table name> are determined as follows:

   a) Case:

      i) If the occurrence of the <table name> (including as a <table name> represented in a <table name with optional column list>) is contained in a <schema>, then let the applicable <authorization identifier> be the <authorization identifier> specified as the <schema authorization identifier> of the <schema>.

      ii) If the occurrence of the <table name> (including as a <table name> represented in a <table name with optional column list>) is contained in an <Ada/SQL DML unit>, then let the applicable <authorization identifier> be the <authorization identifier> implicitly associated with the execution of the program containing the <Ada/SQL DML unit>.

   b) Case:

      i) If the applicable <authorization identifier> is the same as the <authorization identifier> explicitly or implicitly specified in the <table name>, then:

         Case:

            1) If T is a base table, then the applicable <privileges> are INSERT, SELEC, UPDATE, and DELETE, and those <privileges> are grantable.

            2) If T is a viewed table that is not updatable, then the applicable <privileges> are SELEC, and that privilege is grantable if and only if the applicable SELEC privileges on all <table name>s contained in the <query specification> are grantable.

            3) If T is a viewed table that is updatable, then the applicable <privileges> on T are the applicable <privileges> on the <table name> T2 specified in the <from clause> of the <query specification>. A privilege is grantable on T if and only if it is grantable on T2.

      ii) If the applicable <authorization identifier> is not the same as the <authorization identifier> explicitly or implicitly specified in the <table name>, then the applicable <privilege definition>s consist of all <privilege definition>s whose <table name> is the same as the given <table name> and whose <grantee>s either contain the applicable <authorization identifier> or contain PUBLIC, and the applicable <privileges> consist of all <privileges> specified in applicable <privilege definition>s. A privilege is grantable if and only if it is specified in the <privileges>

of some applicable <privilege definition> that specifies WITH_GRANT_OPTION => ENABLED and that specifies the applicable <authorization identifier>.

4) ALL_PRIVILEGES is equivalent to a list of <action>s that include all applicable <privileges> on the <table name>.

5) The applicable <privileges> for the <table name> of a <privilege definition> shall include the <privileges> specified in the <privilege definition>.

General Rules

None.

Notes

1) The Ada/SQL <privilege definition> conforms to the ANSI SQL <privilege definition>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1-SR2	SR1-SR2	-
SR3	SR3	2
SR4	SR4	3
SR5	SR5	-

2) Ada/SQL does not have a construct equivalent to the ANSI SQL <module>. Each ANSI SQL <module> has an explicitly declared <authorization identifier> that is referenced in ANSI SQL SR3a.ii. Each execution of an Ada/SQL program has an implicitly declared <authorization identifier>, which is referenced in Ada/SQL SR3a.ii, and it is this <authorization identifier> that is analogous to that of an ANSI SQL <module>.

3) ANSI SQL SR4 references "ALL" rather than "ALL PRIVILEGES", although the syntax requires "ALL PRIVILEGES". A correction is contemplated for later versions of the ANSI document.

7. Program environment

7.1 <Ada/SQL compilation unit>

Function

Contain the various parts of an Ada/SQL program.

Format

<Ada/SQL compilation unit> ::=
    <authorization package>
  | <schema package declaration>
  | <schema package body>
  | <global variable package>
  | <Ada/SQL DML unit>

Effective Ada Declarations

None.

Example

See examples with various types of <Ada/SQL compilation unit>s.

Syntax Rules

None.

General Rules

None.

Notes

1) An <authorization package> declares an <authorization identifier>. A <schema package declaration> declares data types, subtypes, <named number>s, and tables. A <schema package body> declares unique constraints on views, the <query specification>s defining views, and <privileges>. A <global variable package> declares variables and <correlation name>s that may be referenced from several <Ada/SQL DML unit>s. An <Ada/SQL DML unit> contains <SQL statement>s manipulating a database.

7.2 <Ada/SQL DML unit>

Function

Contain <SQL statement>s manipulating a database.

Format

<Ada/SQL DML unit> ::=
    <context clause>
    <Ada/SQL DML unit header>
    <Ada/SQL DML unit text>
    <Ada/SQL DML unit trailer>

<Ada/SQL DML unit header> ::=
    <procedure header>
  | <function header>
  | <package header>
  | <subunit header>

<subunit header> ::=
    separate <Ada parent unit name> <separate header>

<Ada parent unit name> ::= *legal Ada text not containing any of the following*
            <Ada reserved word>s: **procedure, function, package, task**

<separate header> ::=
    <procedure header>
  | <function header>
  | <package header>
  | <task header>

<procedure header> ::=
    procedure <program identifier>
    [<subprogram specification Ada text>] **is**

<function header> ::=
    function <program identifier> <subprogram specification Ada text> **is**

<package header> ::=
    package body <program identifier> **is**

<task header> ::=
    task body <program identifier> **is**

<subprogram specification Ada text> ::= *legal Ada text not containing the*
                <Ada reserved word> is

<Ada/SQL DML unit trailer> ::=

end [<program identifier>] ;

<Ada/SQL DML unit text> ::=
[<use clause> ...]
[<local variable package>]
[<use clause> ...]
<Ada/SQL embedded text> ...

<Ada/SQL embedded text> ::= *legal Ada text, containing one or more* **<SQL statement>s**

Effective Ada Declarations

None.

Example

```
with EXAMPLE_SDL, EXAMPLE_DML_ADA_SQL;
procedure EXAMPLE_DML is

  package LOCAL_VARIABLES is
    CURSOR : CURSOR_NAME;
  end LOCAL_VARIABLES;
  use LOCAL_VARIABLES;

begin

  DECLAR ( CURSOR , CURSOR_FOR =>
    SELEC      ( '*',
    FROM      => EMPLOYEE ) );
    . . .

  -- process the data

end EXAMPLE_DML;
```

Syntax Rules

1) If both the <Ada/SQL DML unit header> and the <Ada/SQL DML unit trailer> contain <program identifier>s, then both <program identifier>s shall be identical.

2) Let P be the <program identifier> contained in the <Ada/SQL DML unit header>. An <Ada/SQL DML unit> shall contain P_ADA_SQL as a <package name> not contained in a <local variable package> and also as a <library package name> not contained in a <package name>. P_ADA_SQL is called the *Ada/SQL definition package* associated with the <Ada/SQL DML unit>.

General Rules

None.

Notes

1) Section 5.3 contains an SR prohibiting <Ada/SQL embedded text> from containing <Ada/SQL statement name>s, other than the quite common OPEN, CLOSE, and DELETE, except when actually used as part of <SQL statement>s. This simplifies the Ada/SQL automated tools' task of locating <SQL statement>s within <Ada/SQL embedded text>.

2) The restrictions on <Ada reserved word>s contained in <Ada parent unit name>s and <subprogram specification Ada text> are actually not restrictions at all; legal Ada text in those contexts could not contain the forbidden <Ada reserved word>s. The restrictions are stated to make Ada/SQL automated tool parsing strategy obvious, without requiring the tools to understand all of the Ada language.

3) Release 1 implementations do not support <subunit header>s.

4) Release 1 implementations require that each <Ada/SQL DML unit> be contained in a separate source file. This is done so that the Ada/SQL automated tools do not have to use Ada syntax to determine where the <Ada/SQL DML unit> ends.

5) Release 1 implementations do not process <use clause>s or <local variable package>s contained within <Ada/SQL DML unit text>.

7.3 <SQL statement>

Function

Execute operations manipulating a database.

Format

<SQL statement> ::=
  <close statement>
| <commit statement>
| <declare cursor>
| <delete statement: positioned>
| <delete statement: searched>
| <fetch statement>
| <insert statement>
| <open statement>
| <rollback statement>
| <select statement>
| <update statement: positioned>
| <update statement: searched>
| <open database statement>
| <exit database statement>

<open database statement> ::=
  OPEN_DATABASE (<authorization identifier> , <password>) ;

<password> ::=
  <database identifier>

<exit database statement> ::=
  EXIT_DATABASE ;

Effective Ada Declarations

```
procedure OPEN_DATABASE
          ( AUTHORIZATION_IDENTIFIER , PASSWORD : STANDARD.STRING );

procedure EXIT_DATABASE;
```

Example

```
OPEN_DATABASE ( "EXAMPLE" , "HELLO" );

EXIT_DATABASE;
```

see also examples given for each type of <SQL statement>

Syntax Rules

None.

General Rules

1) An <open database statement> shall be the first <SQL statement> executed by a program, otherwise any other <SQL statement> executed before an <open database statement> has been executed merely raises the INVALID_DATABASE_STATE exception. Also, executing a second or subsequent <open database statement> raises the INVALID_DATABASE_STATE exception.

2) The precise meaning of the <open database statement> is implementation-defined.

3) An <exit database statement> shall be the last <SQL statement> executed by a program, otherwise any other <SQL statement> executed after executing an <exit database statement> merely raises the INVALID_DATABASE_STATE exception.

4) When an <SQL statement>, other than a <declare cursor>, <open database statement>, or <exit database statement>, is executed by a program and no transaction is active for the program, a transaction is effectively initiated and associated with the program for this <SQL statement> and subsequent <SQL statement>s executed by the program until the program terminates the transaction.

5) If an <SQL statement> does not execute successfully, then all changes made to the database by the execution of that <SQL statement> are canceled and the appropriate exception (covered by other General Rules) is raised.

Notes

1) <declare cursor> is not an ANSI SQL <SQL statement> because it is not executed in ANSI SQL, as it is in Ada/SQL.

2) The <open database statement> and <exit database statement> are not part of ANSI SQL. They are in Ada/SQL to satisfy requirements of underlying database management systems. The precise semantics of the <open database statement> may vary depending on the particular database management system. For example, in one implementation the <authorization identifier> might specify a particular database to use. With another implementation, the <authorization identifier> might be used to determine privileges for access to a database. There is no requirement that <authorization identifier> and <password> really have the semantic meaning implied by their names; the names are selected based on the most common use.

3) The effective parameters to OPEN_DATABASE are of type STANDARD.STRING to ensure that the type and its component enumeration literals are directly visible without requiring any additional library units to be with'ed by the program.

7.4 <global variable package> and <local variable package>

Function

Declare variables and <correlation name>s to be used in <SQL statement>s.

Format

<global variable package> ::=
    [<context clause>]
    <variable package specification> ;

<local variable package> ::=
    <variable package specification> ;

<variable package specification> ::=
    package <package identifier> is
     [<use clause> ...]
     { <correlation name declaration> | <variable declaration> } ...
    end [<package identifier>]

<variable declaration> ::=
    <simple variable name list> : <subtype indication> ;

<simple variable name list> ::=
    <simple variable name> [{ , <simple variable name> } ...]

<correlation name declaration> ::=
  package <correlation name> is new
   <underscored table name>_CORRELATION.NAME (" <database identifier> ") ;

<underscored table name> ::=
    <table identifier>
  | <authorization identifier>_<table identifier>

Effective Ada Declarations

For a table t with <authorization identifier> a:

```
package a_t_CORRELATION is
  generic
    CORRELATION_NAME : STANDARD.STRING;
  package NAME is
    -- see sections 5.7 and 5.20 for contents
  end NAME;
end a_t_CORRELATION;

package t_CORRELATION renames a_t_CORRELATION;
 -- this declaration is given if and only if the <table identifier> t is
 --  declared in exactly one <schema package> referenced from the
```

```
    --  <variable package specification> for which the declaration is
    --  effective
```

Example

```
with EXAMPLE_TYPES, EXAMPLE_SDL, EXAMPLE_VARIABLES_ADA_SQL;
package EXAMPLE_VARIABLES is

  use EXAMPLE_TYPES.ADA_SQL, EXAMPLE_VARIABLES_ADA_SQL;

  CURRENT_EMPLOYEE,
  HIS_MANAGER       : EMPLOYEE_NAME;
  HIS_SALARY        : EMPLOYEE_SALARY;
  CURSOR            : CURSOR_NAME;
  EMPLOYEE_LAST,
  MANAGER_LAST      : NATURAL;
  SALARY_INDICATOR,
  MANAGER_INDICATOR : INDICATOR_VARIABLE;

  package E is new EMPLOYEE_CORRELATION.NAME ( "E" ); -- employees
  package M is new EMPLOYEE_CORRELATION.NAME ( "M" ); -- managers

end EXAMPLE_VARIABLES;
```

see also section 7.2 for an example of a <local variable package>

Syntax Rules

1) If a <variable package specification> contains two <package identifier>s, then both <package identifier>s shall be identical.

2) Each <simple variable name> and <correlation name> shall be distinct from any other <simple variable name> and <correlation name> within the containing <variable package specification>.

3) If the <type mark> of the <subtype indication> denotes an unconstrained character string subtype, then the <subtype indication> shall also contain an <index constraint> to define the index range of the variable.

4) The <correlation name> and the <database identifier> in a <correlation name declaration> shall be identical.

5) The <table name> represented in an <underscored table name> is determined as follows:

   Case:

   a) If an <authorization identifier> is specified, then the <table name> formed as <authorization identifier>.<table identifier>, using the contained <authorization identifier> and <table identifier>, is the one represented in the <underscored table name>.

b) If an <authorization identifier> is not specified, then the <table name> formed as the contained <table identifier> is the one represented in the <underscored table name>.

6) Let P be the <package identifier> contained in the <variable package specification> contained in a <global variable package>. If and only if the <global variable package> contains any <correlation name declaration>s, then it shall contain P_ADA_SQL as a <package name> and also as a <library package name> not contained in a <package name>. P_ADA_SQL is called the *Ada/SQL definition package* associated with the <global variable package>.

General Rules

1) The first <package identifier> is declared to be the name of the containing <global variable package> or <local variable package>.

2) A <variable declaration> declares its contained <simple variable name>s. Each <simple variable name> denotes a program variable. The subtype of the program variable is given by the <subtype indication> contained in the <variable declaration>.

3) A <correlation name declaration> declares its contained <correlation name> as a <correlation name> for the table whose <table name> is represented in the contained <underscored table name>.

Notes

1) ANSI SQL <correlation name>s are declared in their containing <SQL statement>s. In order to make the effective Ada declarations possible for Ada/SQL, however, it is necessary that Ada/SQL <correlation name>s be separately declared with <correlation name declaration>s. This means that a <correlation name> must always refer to the same table, although it can be used in several different <SQL statement>s.

2) Release 1 implementations require that each <global variable package> be contained in a separate source file, and that the name of the file be the same as the name of the <global variable package>, possibly augmented with an implementation-dependent indication that the file contains Ada source code. This is done so that the text of a <global variable package> can be found when its name is encountered in a <with clause>.

3) Release 1 implementations do not support <local variable package>s.

4) Release 1 implementations do not support <authorization identifier>s in <underscored table name>s.

8. Data manipulation language

General Rules

1) A program shall not, via concurrent tasking, simultaneously execute any of the following statements (including two of the same statement): <delete statement: searched>, <insert statement>, <select statement>, <update statement: searched>. Otherwise, the program is erroneous.

2) A program shall not, via concurrent tasking, simultaneously execute any of the following statements (including two of the same statement) for the same <cursor name>: <close statement>, <declare cursor>, <delete statement: positioned>, <fetch statement>, <open statement>, <update statement: positioned>. Otherwise, the program is erroneous.

3) A program shall not, via concurrent tasking, simultaneously execute a <commit statement> or a <rollback statement> and any statement including a <cursor name> (listed in GR2). Otherwise, the program is erroneous.

4) Any program whose effect depends on the order of processing two statements simultaneously executed (not synchronized by the program) from concurrent tasks is erroneous.

8.1 <close statement>

Function

Close a cursor.

Format

<close statement> ::=
CLOSE (<cursor name>) ;

Effective Ada Declarations

```
procedure CLOSE ( CURSOR : in out CURSOR_NAME );
```

Example

```
CURSOR : CURSOR_NAME;
 . . .
CLOSE ( CURSOR );
```

Syntax Rules

None.

General Rules

1) The program shall have executed a <declare cursor> whose <cursor name> is the same as the <cursor name> of the <close statement>; otherwise, the INVALID_CURSOR_STATE exception is raised.

2) Let CR denote the cursor defined by the last such <declare cursor> executed.

3) Cursor CR shall be in the open state; otherwise, the INVALID_CURSOR_STATE exception is raised.

4) Cursor CR is placed in the closed state and the copy of the <cursor specification> of CR is destroyed.

Notes

1) The Ada/SQL <close statement> conforms to the ANSI SQL <close statement>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL
SR1	GR1-GR2
GR1-GR2	GR3-GR4

2) The expression of ANSI SQL SR1 as Ada/SQL GRs is necessary because a <declare cursor> is declarative in ANSI SQL, while being executable in Ada/SQL.

3) The sequence of operations on a cursor implied by the rules is as follows:

   a) Compile time

      i) Declare a program variable of type CURSOR_NAME to denote the cursor.

   b) Execution time

      i) DECLAR the cursor

      ii) OPEN the cursor

      iii) Process data through the cursor (this aspect is not addressed by the rules of this section)

      iv) CLOSE the cursor

8.2 <commit statement>

Function

Terminate the current transaction with commit.

Format

<commit statement> ::=
COMMIT_WORK ;

Effective Ada Declarations

```
procedure COMMIT_WORK;
```

Example

```
COMMIT_WORK;
```

Syntax Rules

None.

General Rules

1) The current transaction is terminated.

2) Any cursors that were opened by the current transaction are closed.

3) Any changes to the database that were made by the current transaction are committed.

Notes

1) The Ada/SQL <commit statement> conforms to the ANSI SQL <commit statement>.

2) Release 1 implementations do not support the <commit statement>.

8.3 <declare cursor>

Function

Define a cursor.

Format

<declare cursor> ::=
DECLAR (<cursor name> , CURSOR_FOR => <cursor specification>) ;

<cursor specification> ::=
<query expression> [,
<order by clause>]

<query expression> ::=
<query term>
| <query expression> & { UNION | UNION_ALL } (<query term>)
| <query expression> & { UNION | UNION_ALL } <query term>

<query term> ::=
<query specification> | (<query expression>)

<order by clause> ::=
ORDER_BY => <sort specification> [{ & <sort specification> } . . .]

<sort specification> ::=
<sort column specification>
| ASC (<sort column specification>)
| DESC (<sort column specification>)

<sort column specification> ::=
<column number> | <column specification>

<column number> ::=
<integer>

Effective Ada Declarations

```
type QUERY_EXPRESSION is private;

type SORT_SPECIFICATION is private;

type UNIONED_QUERY_EXPRESSION is private;

type COLUMN_NUMBER is range 1 .. implementation_defined;

NULL_SORT_SPECIFICATION : constant SORT_SPECIFICATION;
```

```
procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_EXPRESSION;
            ORDER_BY    : in     SORT_SPECIFICATION
                                 := NULL_SORT_SPECIFICATION );

procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_EXPRESSION;
            ORDER_BY    : in     COLUMN_NUMBER );

procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_EXPRESSION;
            ORDER_BY    : in     COLUMN_SPECIFICATION );

procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_SPECIFICATION;
            ORDER_BY    : in     SORT_SPECIFICATION
                                 := NULL_SORT_SPECIFICATION );

procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_SPECIFICATION;
            ORDER_BY    : in     COLUMN_NUMBER );

procedure DECLAR
          ( CURSOR      : in out CURSOR_NAME;
            CURSOR_FOR  : in     QUERY_SPECIFICATION;
            ORDER_BY    : in     COLUMN_SPECIFICATION );

function "&" ( LEFT  : QUERY_EXPRESSION;
               RIGHT : UNIONED_QUERY_EXPRESSION )
 return QUERY_EXPRESSION;

function "&" ( LEFT  : QUERY_SPECIFICATION;
               RIGHT : UNIONED_QUERY_EXPRESSION )
 return QUERY_EXPRESSION;

function UNION ( RIGHT : QUERY_EXPRESSION ) return UNIONED_QUERY_EXPRESSION;

function UNION_ALL ( RIGHT : QUERY_EXPRESSION )
 return UNIONED_QUERY_EXPRESSION;

function UNION ( RIGHT : QUERY_SPECIFICATION )
 return UNIONED_QUERY_EXPRESSION;

function UNION_ALL ( RIGHT : QUERY_SPECIFICATION )
 return UNIONED_QUERY_EXPRESSION;

function "&" ( LEFT  : SORT_SPECIFICATION;
```

```
                RIGHT : SORT_SPECIFICATION ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_SPECIFICATION;
                RIGHT : COLUMN_SPECIFICATION ) return SORT_SPECIFICATION;

function "&" ( LEFT  : SORT_SPECIFICATION;
                RIGHT : COLUMN_SPECIFICATION ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_SPECIFICATION;
                RIGHT : SORT_SPECIFICATION ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_NUMBER;
                RIGHT : COLUMN_NUMBER ) return SORT_SPECIFICATION;

function "&" ( LEFT  : SORT_SPECIFICATION;
                RIGHT : COLUMN_NUMBER ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_NUMBER;
                RIGHT : SORT_SPECIFICATION ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_SPECIFICATION;
                RIGHT : COLUMN_NUMBER ) return SORT_SPECIFICATION;

function "&" ( LEFT  : COLUMN_NUMBER;
                RIGHT : COLUMN_SPECIFICATION ) return SORT_SPECIFICATION;

function ASC ( RIGHT : COLUMN_SPECIFICATION ) return SORT_SPECIFICATION;

function ASC ( RIGHT : COLUMN_NUMBER ) return SORT_SPECIFICATION;

function DESC ( RIGHT : COLUMN_SPECIFICATION ) return SORT_SPECIFICATION;

function DESC ( RIGHT : COLUMN_NUMBER ) return SORT_SPECIFICATION;
```

Example

```
package E is new EMPLOYEE_CORRELATION.NAME ( "E" ); -- employees
package M is new EMPLOYEE_CORRELATION.NAME ( "M" ); -- managers

CURSOR : CURSOR_NAME;
  . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC     ( NAME & SALARY & MANAGER,
  FROM     => EMPLOYEE ),
  ORDER_BY => MANAGER & DESC(SALARY) );

-- variations: ORDER_BY => 3 & DESC(2)
--             ORDER_BY => ASC(3) & DESC(SALARY), etc.

DECLAR ( CURSOR, CURSOR_FOR =>
      SELEC  ( NAME & SALARY & MANAGER,
      FROM  => EMPLOYEE,
```

```
    WHERE => SALARY > 25_000.00 )
& UNION (
    SELEC  ( E.NAME & E.SALARY & E.MANAGER,
    FROM   => E.EMPLOYEE & M.EMPLOYEE,
    WHERE => EQ ( E.MANAGER , M.NAME )
    AND       E.SALARY > M.SALARY ) ) );

  -- variations: UNION_ALL
```

Syntax Rules

1) A <query term> immediately contained in a <query expression> also immediately containing an ampersand shall be surrounded by parentheses if it immediately contains a <query specification>.

2) Let T denote the table specified by the <cursor specification>.

3) Case:

   a) If ORDER_BY is specified, the T is a read-only table with the specified sort order.

   b) If none of ORDER_BY, UNION, or UNION_ALL is specified and the <query specification> is updatable, then T is an updatable table.

   c) Otherwise, T is a read-only table.

4) Case:

   a) If neither UNION nor UNION_ALL is specified, then the description of T is the description of the <query specification>.

   b) If UNION or UNION_ALL is specified, then for each UNION or UNION_ALL that is specified, let T1 and T2 denote the tables specified by the <query expression> and the <query term>. The <select list>s for the specification of T1 and T2 shall consist of '*', <column specification>s, or <column specification>s to which one or more CONVERT_TO operators are applied.. Except for column names, the descriptions of T1 and T2 shall be identical. All columns of the result are unnamed. Except for <column names>s, the description of the result is the same as the description of T1 and T2.

5) If ORDER_BY is specified, then each <column specification> in the <order by clause> shall identify a column of T, and each <column number> in the <order by clause> shall be greater than 0 and not greater than the degree of T. A named column may be referenced by a <column number> or a <column specification>. An unnamed column shall be referenced by a <column number>.

General Rules

1) The cursor CR denoted by the <cursor name> shall not be in the open state; otherwise, the INVALID_CURSOR_STATE exception is raised.

2) Case:

   a) If T is an updatable table, then the cursor is associated with the named table identified by the <table name> in the <from clause>. Let B denote that named table. For each row in T, there is a corresponding row in B from which the row of T is derived. When the cursor is positioned on a row of T, the cursor is also positioned on the corresponding row of B.

   b) Otherwise, the cursor is not associated with a named table.

3) Case:

   a) If neither UNION nor UNION_ALL is specified, then T is the result of the specified <query specification>.

   b) If UNION or UNION_ALL is specified, then for each UNION or UNION_ALL that is specified let T1 and T2 be the result of the <query expression> and the <query term>. The result of the UNION or UNION_ALL is effectively derived as follows:

      i) Initialize the result to an empty table.

      ii) Insert each row of T1 and each row of T2 into the result.

      iii) If UNION, rather than UNION_ALL, is specified, then eliminate any redundant duplicate rows from the result.

4) Case:

   a) If ORDER_BY is not specified, then the ordering of rows in T is implementor-defined. This order is subject to the reproducibility requirement within a transaction, but it may change between transactions. The execution of a program is erroneous if its effect depends on the ordering of rows in T.

   b) If ORDER_BY is specified, then T has a sort order:

      i) The sort order is a sequence of sort groups. A sort group is a sequence of rows in which all values of a sort column are identical. Furthermore, a sort group may be a sequence of sort groups.

      ii) The cardinality of the sequence and the ordinal position of each sort group is determined by the values of the most significant sort column. The cardinality of the sequence is the minimum number of sort groups such that, for each sort group of more than one row, all values of that sort column are identical.

iii) If the sort order is based on additional sort columns, then each sort group of more than one row is a sequence of sort groups. The cardinality of each sequence and the ordinal position of each sort group within each sequence is determined by the values of the next most significant sort column. The cardinality of each sequence is the minimum number of sort groups such that, for each sort group of more than one row, all values of that sort column are identical.

iv) The preceding paragraph applies in turn to each additional sort column. If a sort group consists of multiple rows and is not a sequence of sort groups, then the order of the rows within that sort group is undefined. The execution of a program is erroneous if its effect depends on the order of rows in such a sort group.

v) Let C be a sort column and let S denote a sequence which is determined by the values of C.

vi) A sort direction is associated with each sort column. If the direction of C is ascending, then the first sort group of S contains the lowest value of C and each successive sort group contains a value of C that is greater than the value of C in its predecessor sort group. If the direction is descending, then the first sort group of S contains the highest value of C and each successive sort group contains a value of C that is less than the value of C in its predecessor sort group.

vii) Ordering is determined by the comparison rules specified in 5.11, "<comparison predicate>". The order of the null value relative to nonnull values is implementor-defined, but shall be either greater than or less than all nonnull values. The execution of a program is erroneous if its effect depends on the order of null values relative to nonnull values.

viii) A <sort specification> specifies a sort column and a direction. The sort column is the column referenced by the <column number> or the <column specification>. The <column number> i references the i-th column of T. A <column specification> references the named column.

ix) If DESC is specified in a <sort specification>, then the direction of the sort column specified by that <sort specification> is descending. If ASC is specified or if neither ASC nor DESC is specified, then the direction of the sort column is ascending.

x) The <sort specification> sequence determines the relative significance of the sort columns. The sort column specified by the first <sort specification> is the most significant sort column and each successively specified sort column is less significant than the previously specified sort column.

5) Let CS denote the <cursor specification>.

6) A copy of CS is effectively created in which each <program object name> is replaced by the value of the entity that it denotes. The execution of a program is erroneous if any such value is undefined.

Notes

1) The only form of <query expression> supported by Release 1 implementations is <query term>.

2) The only form of <query term> supported by Release 1 implementations is <query specification>.

3) The Ada/SQL <declare cursor> conforms to the ANSI SQL <declare cursor>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	4
SR1	—	5
SR2	—	6
SR3-SR5	SR2-SR4	-
SR6	SR5	7
—	GR1	5
GR1-GR3	GR2-GR4	8
—	GR5-GR6	5

4) Without Ada/SQL SR1, the BNF grammar would permit omission of the parentheses around a <query specification> parameter of a UNION or UNION_ALL effective function.

5) <declare cursor>s are declarative within ANSI SQL. Hence, ANSI SQL SR1 ensures that there is only one <declare cursor> for each <cursor name>. <declare cursor>s are, however, executable within Ada/SQL. The same <cursor name> may be used in successive executions of Ada/SQL <declare cursor>s, as long as the cursor denoted by that <cursor name> has not yet been declared or has been closed (see Ada/SQL GR1). Such successive executions may be used to declare several cursors differing only in the values of program objects used as selection criteria. In ANSI SQL, these values are placed in the cursor when the appropriate <open statement> is executed. In Ada/SQL, these values are placed in the cursor when the <declare cursor> is executed (see Ada/SQL GR5-GR6). The Ada/SQL execution of a <declare cursor> followed by an <open statement> for that cursor will be equivalent to the ANSI SQL execution of an <open statement> (using the appropriate program values as parameters) for a corresponding <declare cursor> provided that the Ada/SQL program does not change the values of program variables referenced in the <declare cursor> between execution of the <declare cursor> and its corresponding <open statement>. Note that any potential difference between Ada/SQL and ANSI SQL is limited to the values of program variables; the state of the database is considered to determine which rows will initially be present within the table designated by a cursor at the time the <open statement> is executed, in both Ada/SQL and ANSI SQL.

6) ANSI SQL SR2 is not applicable to Ada/SQL, which does not have <parameter name>s.

7) Ada/SQL uses a unique syntactic class, <column number>, for the number of a column in a <sort specification>, rather than the general ANSI SQL <unsigned integer> (or its Ada/SQL equivalent, <integer>), because of the restriction that a <column number> may not be 0. This restriction is reflected in the effective definition of type COLUMN_NUMBER.

8) Ada/SQL GR4.b.viii corrects two errors in ANSI SQL GR3.b.viii.

8.4 <delete statement: positioned>

Function

Delete a row of a table.

Format

<delete statement: positioned> ::=
DELETE_FROM (<table name> ,
WHERE_CURRENT_OF => <cursor name>) ;
|
DELETE (FROM => <table name> ,
WHERE_CURRENT_OF => <cursor name>) ;

Effective Ada Declarations

```
procedure DELETE_FROM
          ( TABLE             : in     TABLE_NAME;
            WHERE_CURRENT_OF  : in out CURSOR_NAME );

procedure DELETE
          ( FROM              : in     TABLE_NAME;
            WHERE_CURRENT_OF  : in out CURSOR_NAME ) renames DELETE_FROM;
```

Example

```
CURSOR : CURSOR_NAME;
 . . .
DELETE_FROM       ( EMPLOYEE,
WHERE_CURRENT_OF => CURSOR );

-- variation: DELETE    ( FROM => EMPLOYEE,
--            WHERE_CURRENT_OF => CURSOR );
```

Syntax Rules

1) Both forms of the <delete statement: positioned> are equivalent.

2) The applicable <privileges> for the <table name> shall include DELETE.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

General Rules

1) The program shall have executed a <declare cursor> whose <cursor name> is the same as the <cursor name> in the <delete statement: positioned>; otherwise, the INVALID_CURSOR_STATE exception is raised.

2) Let CR denote the cursor defined by the last such <declare cursor> executed.

3) The table designated by CR shall not be a read-only table; otherwise, the INVALID_CURSOR_STATE exception is raised.

4) Let T denote the table identified by the <table name>. T shall be the table identified in the first <from clause> in the <cursor specification> of CR; otherwise, the INVALID_CURSOR_STATE exception is raised.

5) Cursor CR shall be positioned on a row; otherwise, the INVALID_CURSOR_STATE exception is raised.

6) The row from which the current row of CR is derived is deleted.

Notes

1) The Ada/SQL <delete statement: positioned> conforms to the ANSI SQL <delete statement: positioned>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	2
SR1	SR2	-
SR2	GR1-GR2	3
SR3-SR4	GR3-GR4	3
GR1-GR2	GR5-GR6	-

2) The first form of the <delete statement: positioned> is that originally defined for Ada/SQL, and is provided for upward compatibility. The second form is designed to use the same DELETE <key word> as for <privileges>.

3) The expression of ANSI SQL SR2-SR4 as Ada/SQL GRs is necessary because a <declare cursor> is declarative in ANSI SQL, while being executable in Ada/SQL.

4) Release 1 implementations do not support the <delete statement: positioned>.

8.5 <delete statement: searched>

Function

Delete rows of a table.

Format

<delete statement: searched> ::=
  DELETE_FROM (<table name> [,
  WHERE => <search condition>]) ;
|
  DELETE (FROM => <table name> [,
  WHERE => <search condition]) ;

Effective Ada Declarations

```
procedure DELETE_FROM
          ( TABLE : in TABLE_NAME;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure DELETE
          ( FROM  : in TABLE_NAME;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames DELETE_FROM;
```

Example

```
DELETE_FROM ( EMPLOYEE,
WHERE       => SALARY > 25_000.00 );

DELETE_FROM ( EMPLOYEE );

-- variations: DELETE ( FROM => EMPLOYEE,
--             WHERE         => SALARY > 25_000.00 );

--             DELETE ( FROM => EMPLOYEE );
```

Syntax Rules

1) Both forms of the <delete statement: searched> are equivalent.

2) The applicable <privileges> for the <table name> shall include DELETE.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

3) Let T denote the table identified by the <table name>. T shall not be a read-only table or a table that is identified in a <from clause> of any <subquery> contained in the <search condition>.

4) The scope of the <table name> is the entire <delete statement: searched>.

General Rules

1) Case:

   a) If a <search condition> is not specified, then all rows of T are deleted.

   b) If a <search condition> is specified, then it is applied to each row of T with the <table name> bound to that row, and all rows for which the result of the <search condition> is true are deleted. Each <subquery> in the <search condition> is effectively executed for each row of T and the results used in the application of the <search condition> to the given row of T. If any executed <subquery> contains an outer reference to a column of T, the reference is to the value of that column in the given row of T.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

2) If no rows are deleted, the NO_DATA exception is raised.

Notes

1) The Ada/SQL <delete statement: searched> conforms to the ANSI SQL <delete statement: searched>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	2,3
SR1-SR3	SR2-SR4	-
GR1	GR1	-
—	GR2	4

2) The first form of the <delete statement: searched> is that originally defined for Ada/SQL, and is provided for upward compatibility. The second form is designed to use the same DELETE <key word> as for <privileges>.

3) Release 1 implementations do not support the second form of the <delete statement: searched>.

4) Ada/SQL GR2 corresponds to ANSI SQL GR3a.iv of 7.3.

8.6 <fetch statement>

Function

Position a cursor on the next row of a table and retrieve values from that row.

Format

<fetch statement> ::=
FETCH (<cursor name>) ;
<fetch target list>

<fetch target list> ::=
<fetch into substatement>
[<fetch into substatement> ...]

<fetch into substatement> ::=
INTO (<target specification> [, <cursor name>]) ;

Effective Ada Declarations

```
procedure FETCH ( CURSOR : in out CURSOR_NAME );
```

For an integer, floating point, or enumeration program data type ct:

```
procedure INTO
          ( TARGET    : out ct;
            INDICATOR : out INDICATOR_VARIABLE;
            CURSOR    : in  CURSOR_NAME := NULL_CURSOR_NAME );

procedure INTO
          ( TARGET : out ct;
            CURSOR : in  CURSOR_NAME := NULL_CURSOR_NAME );
```

For a character string program data type ct with index data type i:

```
procedure INTO
          ( TARGET    : out ct;
            LAST      : out i;
            INDICATOR : out INDICATOR_VARIABLE;
            CURSOR    : in  CURSOR_NAME := NULL_CURSOR_NAME );

procedure INTO
          ( TARGET    : out ct;
            LAST      : out i;
            CURSOR    : in  CURSOR_NAME := NULL_CURSOR_NAME );
```

Example

```
CURRENT_EMPLOYEE,
```

```
HIS_MANAGER         : EMPLOYEE_NAME;
HIS_SALARY          : EMPLOYEE_SALARY;
CURSOR              : CURSOR_NAME;
EMPLOYEE_LAST,
MANAGER_LAST        : NATURAL;
SALARY_INDICATOR,
MANAGER_INDICATOR   : INDICATOR_VARIABLE;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC ( NAME & SALARY & MANAGER,
  FROM => EMPLOYEE ) );
 . . .
FETCH ( CURSOR );
INTO ( CURRENT_EMPLOYEE , EMPLOYEE_LAST );
INTO ( HIS_SALARY , SALARY_INDICATOR ); -- variation: INTO ( HIS_SALARY );
INTO ( HIS_MANAGER , MANAGER_LAST , MANAGER_INDICATOR );
```

– variations: same as INTO calls shown above, but add last CURSOR parameter

Syntax Rules

1) The <cursor name> of a <fetch into substatement> shall be the same as the <cursor name> of its containing <fetch statement>.

General Rules

1) Any <fetch into substatement> executed by a task while other tasks within the same program are concurrently executing <fetch statement>s shall include a <cursor name>; otherwise, the execution of the program is erroneous.

2) The program shall have executed a <declare cursor> whose <cursor name> is the same as the <cursor name> in the <fetch statement>; otherwise, the INVALID_CURSOR_STATE exception is raised.

3) Let CR denote the cursor defined by the last such <declare cursor> executed.

4) Let T be the table defined by the <cursor specification> of CR.

5) The number of <fetch into substatement>s in the <fetch target list> shall be the same as the degree of table T; otherwise, the INVALID_CURSOR_STATE exception is raised.

6) The data type of the target designated by the <target specification> of the i-th <fetch into substatement> shall be the same as the data type of the i-th column of table T; otherwise, the INVALID_CURSOR_STATE exception is raised.

7) Cursor CR shall be in the open state; otherwise, the INVALID_CURSOR_STATE exception is raised.

8) If the table designated by cursor CR is empty or if the position of CR is on or after the last row, CR is positioned after the last row (if any), the NO_DATA exception is raised, and the values of the targets identified by the <target specification>s of the <fetch into substatement>s are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration target, or if its effect depends on the value of such an undefined character string target.

9) If the position of CR is before a row, CR is positioned on that row and values in that row are assigned to their corresponding targets.

10) If the position of CR is on r, where r is a row other than the last row, CR is positioned on the row immediately after r and values in the row immediately after r are assigned to their corresponding targets.

11) The assignment of values to targets in the <fetch target list> is in an implementor-defined order. The execution of a program is erroneous if its effect depends on this order.

12) If an error occurs during the assignment of a value to a target, then either the DATA_EXCEPTION or the CONSTRAINT_ERROR exception is raised and the values of all targets are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration target, or if its effect depends on the value of such an undefined character string target. (Specific circumstances in which each exception is raised are described below.)

13) Let V be a target and let v denote its corresponding value in the current row of CR.

14) Case:

    a) If v is the null value, then:

       Case:

       i) If an indicator is specified for V, then that indicator is set to NULL_VALUE and the values of the variables denoted by the <variable name>s of the <program variable> and <last variable> (if any) of V are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration variable, or if its effect depends on the value of such an undefined character string variable.

       ii) If an indicator is not specified for V, then the DATA_EXCEPTION exception is raised.

    b) If v is not the null value and V has an indicator, then that indicator is set to NOT_NULL.

15) The target identified by the <target specification> of the i-th <fetch into substatement> in the <fetch target list> corresponds to the i-th value in the current row of CR.

16) If v is not the null value, then:

Case:

a) If V is of a character string type, then:

Case:

i) If the length of v is zero, then:

1) The value of the variable denoted by the <variable name> of the <program variable> of V is left undefined. The execution of a program is erroneous if its effect depends on the value of such an undefined variable.

2) Case:

a. If the index of the first character in the <program variable> of V has a predecessor, then:

Case:

i. If that predecessor belongs to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the value of that variable is set to that predecessor.

ii. If that predecessor does not belong to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the CONSTRAINT_ERROR exception is raised.

b. If the index of the first character in the <program variable> of V does not have a predecessor, then the DATA_EXCEPTION exception is raised.

ii) If the length of v is not zero, and is equal to or less than the length of the <program variable> of V, then:

1. Case:

a. If all characters of v belong to the subtype of the characters of the <program variable> of V, then successive characters of the variable denoted by the <variable name> of the <program variable> are replaced with successive characters of v. Characters not replaced are left undefined; the execution of a program is erroneous if its effect depends on the value of any of these undefined characters.

b. If any characters of v do not belong to the subtype of the characters of the <program variable> of V, then the DATA_EXCEPTION exception

is raised.

2. Case:

   a. If the index of the last character replaced, taken relative to the index bounds of the subtype of the <program variable> of V, belongs to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the value of that variable is set to that index.

   b. If the index of the last character replaced, taken relative to the index bounds of the subtype of the <program variable> of V, does not belong to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the CONSTRAINT_ERROR exception is raised.

iii) If the length of v is greater than the length of the <program variable> of V, then the DATA_EXCEPTION exception is raised.

b) If V is of an integer, floating point, or enumeration type, then:

Case:

i) If v belongs to the subtype of the variable denoted by the <variable name> of the <program variable> of V, then the value of that variable is set to v.

ii) If v does not belong to the subtype of the variable denoted by the <variable name> of the <program variable> of V, then the CONSTRAINT_ERROR exception is raised.

17) If v is not the null value and the column of T from which it is taken is a named column, then the DATA_EXCEPTION exception is raised if v does not belong to the subtype declared for that column.

Notes

1) Release 1 implementations do not support the optional <cursor name> in the <fetch into sub-statement>. Hence, <fetch statement>s may not be issued from more than one concurrently executing task within a program using such an implementation.

2) Release 1 implementations do not support null values. Indicators may therefore not be used within <target specification>s. Only the effective Ada declarations for INTO procedures not including an INDICATOR_VARIABLE parameter are relevant to Release 1 implementations, and the procedures do not include the final CURSOR_NAME parameter.

3) Release 1 implementations have different exceptions for conditions covered by DATA_EXCEPTION and CONSTRAINT_ERROR. They have a separate exception,

NULL_ERROR, which is raised for a null value returned without an indicator variable (DATA_EXCEPTION raised according to this standard). Otherwise, they do not distinguish between DATA_EXCEPTION and CONSTRAINT_ERROR as does this standard; CONSTRAINT_ERROR is raised in all cases. Due to this standard's lack of a separate NULL_ERROR, application programs can no longer explicitly distinguish errors that would have raised NULL_ERROR. Enhanced error reporting is planned for later versions of SQL, and incorporation of those features into Ada/SQL should restore this capability.

4) The Ada/SQL <fetch statement> conforms to the ANSI SQL <fetch statement>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	5
—	GR1	5
SR1-SR2	GR2-GR5	6
SR3	GR6	6,7
GR1-GR7	GR7-GR13	-
GR8	GR14	8
GR9	GR15	-
GR10	GR16	9
—	GR17	10

5) If <fetch into substatement>s are actually implemented by their effective Ada declarations, then each INTO procedure executed must somehow be associated with its appropriate FETCH. If only a single task in a program is issuing FETCHes and INTOs, then the association can be maintained with a global variable. If, however, several concurrently executing tasks in the same program are issuing FETCHes and INTOs, the <cursor name> mechanism provides the association.

6) The expression of ANSI SQL SR1-SR3 as Ada/SQL GRs is necessary because a <declare cursor> is declarative in ANSI SQL, while being executable in Ada/SQL.

7) Ada/SQL GR6 expresses one aspect of Ada/SQL's strong typing. Release 1 implementations do not provide this aspect of type checking. CONSTRAINT_ERROR may be raised for gross typing violations, such as retrieving a character string column into an integer variable.

8) Ada/SQL indicator variables are of an enumeration type, used only to indicate whether or not a value is null. The condition for which ANSI SQL indicator variables serve a dual purpose as flags, truncation of a retrieved character string, is an error in Ada/SQL.

9) See the general discussion of character strings in section 4.2.1. Ada/SQL does not right pad retrieved character strings with blanks, to be analogous to TEXT_IO. It is considered an error to retrieve a character string value that is longer than the target variable, consistent with Ada's constraint checking for array assignment. Ada/SQL retrievals provide subtype checking.

10) Ada/SQL validates subtype constraints when data are placed into a database by a program, and also validates them when data are retrieved from a database. Unless the database also supports such constraint checking, however, it is possible to execute database operations wherein data violating subtype constraints are created within the database, without passing through a program. GR17 provides for treating the retrieval of such invalid data as an error condition. Release 1 implementations do not support this checking, although they do, of course, perform the subtype checking of GR16, since that is a consequence of Ada semantics. If values from a column are always retrieved into variables of the same subtype as the column, then this Ada constraint checking is equivalent to that of GR17.

8.7 <insert statement>

Function

Create new rows in a table.

Format

<insert statement> ::=
  INSERT_INTO (<table name with optional column list> ,
  { VALUES <= <insert value list> } | <query specification>) ;
|
  INSERT (INTO => <table name with optional column list> ,
  { VALUES <= <insert value list> } | <query specification>) ;

<insert value list> ::=
  <insert value> [{ and <insert value> } ...]

<insert value> ::=
  <value specification> | NULL_VALUE

Effective Ada Declarations

```
type INSERT_VALUE_LIST is private;

type INSERT_VALUE_LIST_STARTER is private;

type NULL_INSERT_VALUE is private;

procedure INSERT_INTO
          ( TABLE  : in TABLE_NAME;
            VALUES : in INSERT_VALUE_LIST );

procedure INSERT_INTO
          ( TABLE  : in TABLE_NAME_WITH_COLUMN_LIST;
            VALUES : in INSERT_VALUE_LIST );

procedure INSERT_INTO
          ( TABLE : in TABLE_NAME;
            QUERY : in QUERY_SPECIFICATION );

procedure INSERT_INTO
          ( TABLE : in TABLE_NAME_WITH_COLUMN_LIST;
            QUERY : in QUERY_SPECIFICATION );

procedure INSERT
          ( INTO   : in TABLE_NAME;
            VALUES : in INSERT_VALUE_LIST ) renames INSERT_INTO;

procedure INSERT
```

```
          ( INTO   : in TABLE_NAME_WITH_COLUMN_LIST;
            VALUES : in INSERT_VALUE_LIST ) renames INSERT_INTO;

  procedure INSERT
          ( INTO  : in TABLE_NAME;
            QUERY : in QUERY_SPECIFICATION ) renames INSERT_INTO;

  procedure INSERT
          ( INTO  : in TABLE_NAME_WITH_COLUMN_LIST;
            QUERY : in QUERY_SPECIFICATION ) renames INSERT_INTO;

  function VALUES return INSERT_VALUE_LIST_STARTER;

  function NULL_VALUE return NULL_INSERT_VALUE;

  function "<="
          ( LEFT  : INSERT_VALUE_LIST_STARTER;
            RIGHT : VALUE_SPECIFICATION ) return INSERT_VALUE_LIST;

  function "<="
          ( LEFT  : INSERT_VALUE_LIST_STARTER;
            RIGHT : NULL_INSERT_VALUE ) return INSERT_VALUE_LIST;

  function "and"
          ( LEFT  : INSERT_VALUE_LIST;
            RIGHT : VALUE_SPECIFICATION ) return INSERT_VALUE_LIST;

  function "and"
          ( LEFT  : INSERT_VALUE_LIST;
            RIGHT : NULL_INSERT_VALUE ) return INSERT_VALUE_LIST;
```

For a program data type ct:

```
  function "<="
          ( LEFT  : INSERT_VALUE_LIST_STARTER;
            RIGHT : ct ) return INSERT_VALUE_LIST;

  function "and"
          ( LEFT  : INSERT_VALUE_LIST;
            RIGHT : ct ) return INSERT_VALUE_LIST;
```

Example

```
NEW_EMPLOYEE,
HIS_MANAGER : EMPLOYEE_NAME;
HIS_SALARY  : EMPLOYEE_SALARY;
 . . .
INSERT_INTO ( EMPLOYEE ( NAME & SALARY & MANAGER ) ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
 . . .
INSERT_INTO ( EMPLOYEE ,
  SELEC ( '*',
```

```
  FROM => NEW_EMPLOYEE_FILE ) );

-- assume NEW_EMPLOYEE_FILE is another database table structured identically
-- to the EMPLOYEE table

-- variations:

INSERT ( INTO => EMPLOYEE ( NAME & SALARY & MANAGER ) ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );
 . . .
INSERT ( INTO => EMPLOYEE ,
  SELEC ( '*',
  FROM => NEW_EMPLOYEE_FILE ) );
```

Syntax Rules

1) Both forms of the <insert statement> are equivalent.

2) The applicable <privileges> for the <table name> represented in the <table name with optional column list> shall include INSERT.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

3) Let T denote the table identified by the <table name> represented in the <table name with optional column list>. T shall not be a read-only table or a table that is identified in a <from clause> of the <query specification> or of any <subquery> contained in the <query specification>.

4) Each <column name> in the <column list> (if any) of the <table name with optional column list> shall identify a column of T and the same column shall not be identified more than once. Omission of the <column list> from the <table name with optional column list> is an implicit specification of a <column list> that identifies all columns of T in the ascending sequence of their ordinal position within T.

5) A column identified by the <column list> of the <table name with optional column list> is an *object column*.

6) Case:

   a) If an <insert value list> is specified, then the number of <insert value>s in that <insert value list> shall be equal to the number of <column name>s in the <column list>. Let the i-th item of the <insert statement> refer to the i-th <insert value> in that <insert value list>.

   b) If a <query specification> is specified, then the degree of the table specified by that <query specification> shall be equal to the number of <column name>s in the <column list>. Let the i-th item of the <insert statement> refer to the i-th column of the table

specified by the <query specification>.

7) If the i-th item of the <insert statement> is not the <insert value> NULL_VALUE, then the data type of the column of table T designated by the i-th <column name> shall be the same as the data type of the i-th item of the <insert statement>.

General Rules

1) A row is inserted in the following steps:

   a) A candidate row is effectively created in which the value of each column is the null value. If T is a base table, B, then the candidate row includes a null value for every column of B. If T is a viewed table, the candidate row includes a null value for every column of the base table, B, from which T is derived.

   b) For each object column in the candidate row, the value is replaced by an insert value.

   c) The candidate row is inserted in B.

2) If T is a viewed table defined by a <view definition> that specifies "WITH_CHECK_OPTION", and the <query specification> contained in the <view definition> specifies a <where clause>, then the <search condition> of that <where clause> shall be true for the candidate row; otherwise, the CONSTRAINT_VIOLATION exception is raised.

3) If an <insert value list> is specified, then:

   Case:

   a) If the i-th <insert value> of the <insert value list> is a <value specification>, then the value of the column of the candidate row corresponding with the i-th object column is the value of that <value specification>.

   b) If the i-th <insert value> of the <insert value list> is NULL_VALUE, then the value of the column of the candidate row corresponding with the i-th object column is the null value.

4) If a <query specification> is specified, let R be the result of the <query specification>. If R is empty, then the NO_DATA exception is raised and no row is inserted. The number of candidate rows created is equal to the cardinality of R. The insert values of one candidate row are the values in one row of R and the values in one row of R are the insert values of one candidate row.

5) Let V denote a row of R or the sequence of values specified by the <insert value list>. The i-th value of V is the insert value of the object column identified by the i-th <column name> in the <table name with optional column list>.

6) Let C denote an object column. Let v denote a nonnull insert value of C.

7) Case:

a) If C is of a character string data type and any character of v does not belong to the subtype of the characters in C, then:

Case:

i) If an <insert value list> is specified, then the DATA_EXCEPTION exception is raised.

ii) If a <query specification> is specified, then the program executing the <insert statement> is erroneous.

b) If C is of a character string data type and the length of v is equal to the maximum number of characters C can contain, then the value of C is set to v.

c) If C is of a character string data type and can contain a maximum of L characters, and the length M of v is smaller than L, then the first M characters of C are set to v, and the last L-M characters of C are set to the space character.

d) If C is of a character string data type and the length of v is greater than the maximum number of characters C can contain, then:

Case:

i) If an <insert value list> is specified, then the DATA_EXCEPTION exception is raised.

ii) If a <query specification> is specified, then the program executing the <insert statement> is erroneous.

e) If C is of an integer, floating point, or enumeration data type, and v belongs to the subtype of C, then the value of C is set to v.

f) If C is of an integer, floating point, or enumeration data type, and v does not belong to the subtype of C, then:

Case:

i) If an <insert value list> is specified, then the DATA_EXCEPTION exception is raised.

ii) If a <query specification> is specified, then the program executing the <insert statement> is erroneous.

Notes

1) Release 1 implementations do not support the NULL_VALUE <insert value>.

2) The Ada/SQL <insert statement> conforms to the ANSI SQL <insert statement>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	3
SR1-SR5	SR2-SR6	-
SR6	SR7	4
GR1	GR1	-
GR2	GR2	5
GR3-GR6	GR3-GR6	-
GR7	GR7	6

3) The first form of the <insert statement> is that originally defined for Ada/SQL, and is provided for upward compatibility. The second form is designed to use the same INSERT <key word> as for <privileges>.

4) SR7.a expresses one aspect of Ada/SQL's strong typing. The length restriction of ANSI SQL SR6.a is covered as Ada/SQL GR7.d.

5) ANSI SQL GR2 qualifies the <where clause> of the <view definition> as "not contained in a <subquery>". A <subquery> cannot appear in a <view definition> to which GR2 applies, however. T must be an updatable table, and it is a consequence of other rules that a <subquery> cannot be used in the definition of an updatable table. The detection and/or reporting of check violations is not standardized in Release 1 implementations.

6) GR7 expresses Ada/SQL's subtype checking on data inserted into a database. Release 1 implementations do not support and/or standardize this subtype checking. However, if all insert operations are done using corresponding columns and program variables/values of the same subtype, then Ada's subtype checking prevents violation of Ada/SQL subtype constraints for data inserted from a program. Release 1 implementations do support the strong type checking expressed in SR7.

   Unless the database supports subtype checking, it is possible to create data, not passing through a program, that violate subtype constraints. This may be done with a <query specification> used to create rows of data. For this reason, a program creating such data is considered erroneous. Data inserted from a program, i.e., with an <insert value list> specified, can be checked by the Ada/SQL system, so the DATA_EXCEPTION exception can be raised. An implementation

that can support database subtype checking may raise the DATA_EXCEPTION exception upon detecting a subtype constraint violation.

7) See 5.26, <table name with optional column list>, for notes on its use in Ada/SQL syntax.

8.8 <open statement>

Function

Open a cursor.

Format

<open statement> ::=
OPEN (<cursor name>) ;

Effective Ada Declarations

```
procedure OPEN ( CURSOR : in out CURSOR_NAME );
```

Example

```
CURSOR : CURSOR_NAME;
. . .
OPEN ( CURSOR );
```

Syntax Rules

None.

General Rules

1) The program shall have executed a <declare cursor> whose <cursor name> is the same as the <cursor name> of the <open statement>; otherwise, the INVALID_CURSOR_STATE exception is raised.

2) Let CR denote the cursor defined by the last such <declare cursor> executed.

3) Cursor CR shall be in the closed state; otherwise, the INVALID_CURSOR_STATE exception is raised.

4) Let S denote the <cursor specification> of cursor CR.

5) Cursor CR is opened in the following steps:

   a) If S specifies a read-only table, then that table, as specified by the copy of S made when the <declare cursor> was executed (see 8.3, GR6), is effectively created.

   b) Cursor CR is placed in the open state and its position is before the first row of the table.

Notes

1) The Ada/SQL <open statement> conforms to the ANSI SQL <open statement>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	GR1-GR2	2
GR1-GR2	GR3-GR4	-
GR3	GR5	3

2) The expression of ANSI SQL SR1 as Ada/SQL GRs is necessary because a <declare cursor> is declarative in ANSI SQL, while being executable in Ada/SQL.

3) See also note 5, section 8.3. In Ada/SQL, the evaluation of program values used in cursors occurs at <declare cursor> time; while occurring at <open statement> time in ANSI SQL.

8.9 <rollback statement>

Function

Terminate the current transaction with rollback.

Format

<rollback statement> ::=
ROLLBACK_WORK ;

Effective Ada Declarations

```
procedure ROLLBACK_WORK;
```

Example

```
ROLLBACK_WORK;
```

Syntax Rules

None.

General Rules

1) Any changes to the database that were made by the current transaction are canceled.

2) Any cursors that were opened by the current transaction are closed.

3) The current transaction is terminated.

Notes

1) The Ada/SQL <rollback statement> conforms to the ANSI SQL <rollback statement>.

2) Release 1 implementations do not support the <rollback statement>.

8.10 <select statement>

Function

Retrieve values from a specified row of a table.

Format

<select statement> ::=
[SELEC | SELECT_ALL | SELECT_DISTINCT | SELEC_ALL | SELEC_DISTINCT]
(<select list> ,
<table expression>);
<select target list>

<select target list> ::=
<select into substatement>
[<select into substatement> ...]

<select into substatement> ::=
INTO (<target specification>) ;

Effective Ada Declarations

```
type STAR_TYPE is ( '*' );

procedure SELEC
          ( WHAT  : in SELECT_LIST;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELEC
          ( WHAT  : in VALUE_EXPRESSION;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELEC
          ( WHAT  : in STAR_TYPE;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELECT_ALL
          ( WHAT  : in SELECT_LIST;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELECT_ALL
          ( WHAT  : in VALUE_EXPRESSION;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );
```

```
procedure SELECT_ALL
          ( WHAT  : in STAR_TYPE;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELECT_DISTINCT
          ( WHAT  : in SELECT_LIST;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELECT_DISTINCT
          ( WHAT  : in VALUE_EXPRESSION;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELECT_DISTINCT
          ( WHAT  : in STAR_TYPE;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

procedure SELEC_ALL
          ( WHAT  : in SELECT_LIST;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames SELECT_ALL;

procedure SELEC_ALL
          ( WHAT  : in VALUE_EXPRESSION;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames SELECT_ALL;

procedure SELEC_ALL
          ( WHAT  : in STAR_TYPE;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames SELECT_ALL;

procedure SELEC_DISTINCT
          ( WHAT  : in SELECT_LIST;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames SELECT_DISTINCT;

procedure SELEC_DISTINCT
          ( WHAT  : in VALUE_EXPRESSION;
            FROM  : in FROM_CLAUSE;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
 renames SELECT_DISTINCT;

procedure SELEC_DISTINCT
          ( WHAT  : in STAR_TYPE;
```

```
            FROM   : in FROM_CLAUSE;
            WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
  renames SELECT_DISTINCT;
```

For a program data type ct:

```
  procedure SELEC
            ( WHAT   : in ct;
              FROM   : in FROM_CLAUSE;
              WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

  procedure SELECT_ALL
            ( WHAT   : in ct;
              FROM   : in FROM_CLAUSE;
              WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

  procedure SELECT_DISTINCT
            ( WHAT   : in ct;
              FROM   : in FROM_CLAUSE;
              WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );

  procedure SELEC_ALL
            ( WHAT   : in ct;
              FROM   : in FROM_CLAUSE;
              WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
   renames SELECT_ALL;

  procedure SELEC_DISTINCT
            ( WHAT   : in ct;
              FROM   : in FROM_CLAUSE;
              WHERE  : in SEARCH_CONDITION := NULL_SEARCH_CONDITION )
   renames SELECT_DISTINCT;
```

The INTO procedures shown in section 8.6, effectively declared for <fetch into substatement>s, are also applicable to <select into substatement>s. When one of these INTO procedures is effectively used for a <select into substatement>, however, the final CURSOR parameter may not be specified, according to the syntax rules.

Example

```
DESIRED_EMPLOYEE,
HIS_MANAGER        : EMPLOYEE_NAME;
HIS_SALARY         : EMPLOYEE_SALARY;
EMPLOYEE_LAST,
MANAGER_LAST       : NATURAL;
SALARY_INDICATOR,
MANAGER_INDICATOR  : INDICATOR_VARIABLE;
 . . .
SELEC  ( SALARY & MANAGER ,                  -- variations: SELECT_ALL
FROM  => EMPLOYEE,                           --             SELECT_DISTINCT
WHERE => EQ ( NAME , DESIRED_EMPLOYEE ) ); --             SELEC_ALL
INTO ( HIS_SALARY );                         --             SELEC_DISTINCT
```

```
INTO ( HIS_MANAGER , MANAGER_LAST );
 . . .
SELEC  ( NAME & SALARY & MANAGER ,          -- variations: SELECT_ALL
FROM  => ONE_EMPLOYEE_TABLE );              --             SELECT_DISTINCT
INTO ( DESIRED_EMPLOYEE , EMPLOYEE_LAST ); --             SELEC_ALL
INTO ( HIS_SALARY , SALARY_INDICATOR );     --             SELEC_DISTINCT
INTO ( HIS_MANAGER , MANAGER_LAST , MANAGER_INDICATOR );

-- assume ONE_EMPLOYEE_TABLE is another database table structured
-- identically to the EMPLOYEE table, but containing only one row
```

Syntax Rules

1) Specifying SELEC_ALL is equivalent to specifying SELECT_ALL; specifying SELEC_DISTINCT is equivalent to specifying SELECT_DISTINCT.

2) The applicable <privileges> for each <table name> contained in the <table expression> shall include SELEC.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

3) The <table expression> shall not include a <group by clause> or a <having clause> and shall not identify a grouped view.

4) The number of elements in the <select list> shall be the same as the number of elements in the <select target list>.

5) The data type of the target designated by the <target specification> of the i-th <select into substatement> shall be the same as the data type of the i-th <value expression> in the <select list>.

General Rules

1) Let S be a <query specification> whose <select list> and <table expression> are those specified in the <select statement> and which specifies SELECT_ALL or SELECT_DISTINCT (or their equivalents) if it is specified in the <select statement>. Let R denote the result of <query specification> S.

2) The cardinality of R shall not be greater than one; otherwise, the CARDINALITY_VIOLATION exception is raised. If R is empty, then the NO_DATA exception is raised. In either case, the values of the targets identified by the <target specification>s of the <select into substatement>s are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration target, or if its effect depends on the value of such an undefined character string target.

3) If R is not empty, then values in the row of R are assigned to their corresponding targets.

4) The assignment of values to targets in the <select target list> is in an implementor-defined order. The execution of a program is erroneous if its effect depends on this order.

5) If an error occurs during the assignment of a value to a target, then either the DATA_EXCEPTION or the CONSTRAINT_ERROR exception is raised and the values of all targets are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration target, or if its effect depends on the value of such an undefined character string target. (Specific circumstances in which each exception is raised are described below.)

6) The target identified by the <target specification> of the i-th <select into substatement> in the <select target list> corresponds to the i-th value in the row of R.

7) Let V be an identified target and let v denote its corresponding value in the row of R.

8) Case:

   a) If v is the null value, then:

      Case:

      i) If an indicator is specified for V, then that indicator is set to NULL_VALUE and the values of the variables denoted by the <variable name>s of the <program variable> and <last variable> (if any) of V are undefined. The execution of a program is erroneous if it attempts to evaluate such an undefined integer, floating point, or enumeration variable, or if its effect depends on the value of such an undefined character string variable.

      ii) If an indicator is not specified for V, then the DATA_EXCEPTION exception is raised.

   b) If v is not the null value and V has an indicator, then that indicator is set to NOT_NULL.

9) If v is not the null value, then:

   Case:

   a) If V is of a character string data type, then:

      Case:

      i) If the length of v is zero, then:

         1. The value of the variable denoted by the <variable name> of the <program variable> of V is left undefined. The execution of a program is erroneous if its effect depends on the value of such an undefined variable.

2. Case:

   a. If the index of the first character in the <program variable> of V has a predecessor, then:

      Case:

      i. If that predecessor belongs to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the value of that variable is set to that predecessor.

      ii. If that predecessor does not belong to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the CONSTRAINT_ERROR exception is raised.

   b. If the index of the first character in the <program variable> of V does not have a predecessor, then the DATA_EXCEPTION exception is raised.

ii) If the length of v is not zero, and is equal to or less than the length of the <program variable> of V, then:

1. Case:

   a. If all characters of v belong to the subtype of the characters of the <program variable> of V, then successive characters of the variable denoted by the <variable name> of the <program variable> are replaced with successive characters of v. Characters not replaced are left undefined; the execution of a program is erroneous if its effect depends on the value of any of these undefined characters.

   b. If any characters of v do not belong to the subtype of the characters of the <program variable> of V, then the DATA_EXCEPTION exception is raised.

2. Case:

   a. If the index of the last character replaced, taken relative to the index bounds of the subtype of the <program variable> of V, belongs to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the value of that variable is set to that index.

   b. If the index of the last character replaced, taken relative to the index bounds of the subtype of the <program variable> of V, does not belong to the subtype of the variable denoted by the <variable name> of the <last variable> of V, then the CONSTRAINT_ERROR exception is

raised.

iii) If the length of v is greater than the length of the <program variable> of V, then the DATA_EXCEPTION exception is raised.

b) If V is of an integer, floating point, or enumeration data type, then:

Case:

i) If v belongs to the subtype of the variable denoted by the <variable name> of the <program variable> of V, then the value of that variable is set to v.

ii) If v does not belong to the subtype of the variable denoted by the <variable name> of the <program variable> of V, then the CONSTRAINT_ERROR exception is raised.

10) If v is not the null value and the column of R from which it is taken is a named column, then the DATA_EXCEPTION exception is raised if v does not belong to the subtype declared for that column.

Notes

1) There are four <select statement> procedures effectively declared for each <key word> that may be used to introduce the statement. The procedures differ in the type of their first parameter, based on the text of the <select statement> as follows:

SELECT_LIST - used when the <select list> contains more than one <value expression>

VALUE_EXPRESSION - used when the <select list> contains only one <value expression>, which contains at least one of a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

t (program type) - used when the <select list> contains only one <value expression>, which does not contain a <column specification>, a <set function specification>, an <indicator specification>, or the <key word> USER

STAR_TYPE - used when the <select list> consists of the single element '*'

2) The procedures effectively declared for the <select statement> do not require GROUP_BY or HAVING parameters (unlike the similar functions effectively declared for <subquery>s and <query specification>s) because SR3 prohibits <group by clause>s and <having clause>s from being used in <select statement>s.

3) Release 1 implementations do not support null values. Indicators may therefore not be used within <target specification>s. Only the effective Ada declarations for INTO procedures not including an INDICATOR_VARIABLE parameter (see section 8.6) are relevant to Release 1 implementations. As already noted, the final CURSOR_NAME parameter on the INTO procedures is not relevant to <select statement>s.

4) Release 1 implementations have different exceptions for conditions covered by DATA_EXCEPTION and CONSTRAINT_ERROR. They have a separate exception, NULL_ERROR, which is raised for a null value returned without an indicator variable (DATA_EXCEPTION raised according to this standard). Otherwise, they do not distinguish between DATA_EXCEPTION and CONSTRAINT_ERROR as does this standard; CONSTRAINT_ERROR is raised in all cases. Due to this standard's lack of a separate NULL_ERROR, application programs can no longer explicitly distinguish errors that would have raised NULL_ERROR. Enhanced error reporting is planned for later versions of SQL, and incorporation of those features into Ada/SQL should restore this capability.

5) The Ada/SQL <select statement> conforms to the ANSI SQL <select statement>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
—	SR1	6
SR1-SR3	SR2-SR4	-
SR4	SR5	7
GR1-GR7	GR1-GR7	-
GR8	GR8	8
GR9	GR9	9
—	GR10	10

6) The <key word>s SELECT_ALL and SELECT_DISTINCT are those originally defined for Ada/SQL, and are provided for upward compatibility. SELEC_ALL and SELEC_DISTINCT are provided to use the same SELEC keyword as for <privileges> and because some users have expressed a preference for them. Release 1 implementations do not recognize the new <key word>s.

7) SR5 expresses one aspect of Ada/SQL's strong typing.

8) Ada/SQL indicator variables are of an enumeration type, used only to indicate whether or not a value is null. The condition for which ANSI SQL indicator variables serve a dual purpose as flags, truncation of a retrieved character string, is an error in Ada/SQL.

9) See the general discussion of character strings in section 4.2.1. Ada/SQL does not right pad retrieved character strings with blanks, to be analogous to TEXT_IO. It is considered an error to retrieve a character string value that is longer than the target variable, consistent with Ada's constraint checking for array assignment. Ada/SQL retrievals provide subtype checking.

10) Ada/SQL validates subtype constraints when data are placed into a database by a program, and also validates them when data are retrieved from a database. Unless the database also supports such constraint checking, however, it is possible to execute database operations wherein data violating subtype constraints are created within the database, without passing

through a program. GR10 provides for treating the retrieval of such invalid data as an error condition. Release 1 implementations do not support this checking, although they do, of course, perform the subtype checking of GR9, since that is a consequence of Ada semantics. If values from a column are always retrieved into variables of the same subtype as the column, then this Ada constraint checking is equivalent to that of GR10.

8.11 <update statement: positioned>

Function

Update a row of a table.

Format

<update statement: positioned> ::=
UPDATE (<table name> ,
SET => <set clause: positioned>
[{ and <set clause: positioned> } ...] ,
WHERE_CURRENT_OF => <cursor name>) ;

<set clause: positioned> ::=
<object column: positioned> <= { <value expression> | NULL_VALUE }

<object column: positioned> ::= <column name>

Effective Ada Declarations

For a table t:

```
type SET_CLAUSE_t is private;

procedure UPDATE
          ( TABLE             => in     TABLE_NAME_t;
            SET               => in     SET_CLAUSE_t;
            WHERE_CURRENT_OF => in out CURSOR_NAME );

function "and" ( LEFT , RIGHT : SET_CLAUSE_t ) return SET_CLAUSE_t;

For a column of t of data type ct:

  function "<="
           ( LEFT  : COLUMN_NAME_t_ct;
             RIGHT : VALUE_EXPRESSION_ct ) return SET_CLAUSE_t;

  function "<="
           ( LEFT  : COLUMN_NAME_t_ct;
             RIGHT : ct ) return SET_CLAUSE_t;

  function "<="
           ( LEFT  : COLUMN_NAME_t;
             RIGHT : NULL_INSERT_VALUE ) return SET_CLAUSE_t;
```

NOTE: These effective Ada declarations, other than for the UPDATE procedure, also pertain to 8.12.

Example

```
CURSOR : CURSOR_NAME;
 . . .
UPDATE ( EMPLOYEE ,
SET    => SALARY  <= 2.0 * SALARY
       and MANAGER <= NULL_VALUE ,
WHERE_CURRENT_OF => CURSOR );
 . . .
UPDATE ( EMPLOYEE ,
SET    => SALARY <= 0.0 ,
WHERE_CURRENT_OF => CURSOR );
```

Syntax Rules

1) The applicable <privileges> for the <table name> shall include UPDATE for each <object column: positioned>.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

2) Let T denote the table identified by the <table name>.

3) A <value expression> in a <set clause: positioned> shall not include a <set function specification>.

4) Each <column name> specified as an <object column: positioned> shall identify a column of T. The same <object column: positioned> shall not appear more than once in an <update statement: positioned>.

5) The scope of the <table name> is the entire <update statement: positioned>.

6) For each <set clause: positioned>:

   Case:

   a) If NULL_VALUE is specified, then the column designated by the <object column: positioned> shall allow nulls.

   b) If NULL_VALUE is not specified, then the data type of the column designated by the <object column: positioned> shall be the same as the data type of the <value expression>.

General Rules

1) The program shall have executed a <declare cursor> whose <cursor name> is the same as the <cursor name> in the <update statement: positioned>; otherwise, the INVALID_CURSOR_STATE exception is raised.

2) Let CR denote the cursor defined by the last such <declare cursor> executed.

3) The table designated by CR shall not be a read-only table; otherwise, the INVALID_CURSOR_STATE exception is raised.

4) T shall be the table identified in the first <from clause> in the <cursor specification> of CR; otherwise, the INVALID_CURSOR_STATE exception is raised.

5) Cursor CR shall be positioned on a row; otherwise, the INVALID_CURSOR_STATE exception is raised.

6) The object row is that row from which the current row of CR is derived.

7) The object row is updated as specified by each <set clause: positioned>. A <set clause: positioned> specifies an object column and an update value of that column. The object column is the column identified by the <object column: positioned> in the <set clause: positioned>. The update value is the null value or the value specified by the <value expression>. If the <value expression> contains a reference to a column of T, the reference is to the value of that column in the object row before any value of the object row is updated.

8) The object row is updated in the following steps:

   a) A candidate row is created which is a copy of the object row.

   b) For each <set clause: positioned>, the value of the specified object column in the candidate row is replaced by the specified update value.

   c) The object row is replaced by the candidate row.

9) If T is a viewed table defined by a <view definition> that specifies "WITH_CHECK_OPTION", and the <query specification> contained in the <view definition> specifies a <where clause>, then the <search condition> of that <where clause> shall be true for the candidate row; otherwise, the CONSTRAINT_VIOLATION exception is raised.

10) Let C denote an object column. Let v denote a nonnull update value of C.

11) Case:

   a) If C is of a character string data type and any character of v does not belong to the subtype of the characters in C, then the program executing the <update statement: positioned> is erroneous.

   b) If C is of a character string data type and the length of v is equal to the maximum number of characters C can contain, then the value of C is set to v.

c) If C is of a character string data type and can contain a maximum of L characters, and the length M of v is smaller than L, then the first M characters of C are set to v, and the last L-M characters of C are set to the space character.

d) If C is of a character string data type and the length of v is greater than the maximum number of characters C can contain, then the program executing the <update statement: positioned> is erroneous.

e) If C is of an integer, floating point, or enumeration data type, and v belongs to the subtype of C, then the value of C is set to v.

f) If C is of an integer, floating point, or enumeration data type, and v does not belong to the subtype of C, then the program executing the <update statement: positioned> is erroneous.

Notes

1) The Ada/SQL <update statement: positioned> conforms to the ANSI SQL <update statement: positioned>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1	SR1	-
SR2-SR3	GR1-GR3	2
SR4	SR2,GR4	2
SR5-SR7	SR3-SR5	-
SR8	SR6	3
GR1-GR4	GR5-GR8	-
GR5	GR9	4
GR6	GR10	-
GR7	GR11	5

2) The expression of ANSI SQL SR2-SR4 as Ada/SQL GRs is necessary because a <declare cursor> is declarative in ANSI SQL, while being executable in Ada/SQL.

3) SR6 expresses one aspect of Ada/SQL's strong typing. The length restriction of ANSI SQL SR8.b is covered as Ada/SQL GR11.d.

4) ANSI SQL GR5 qualifies the <where clause> of the <view definition> as "not contained in a <subquery>". A <subquery> cannot appear in a <view definition> to which GR5 applies, however. The table designated by CR must be an updatable table, and it is a consequence of other rules that a <subquery> cannot be used in the definition of an updatable table. The detection and/or reporting of check violations is not standardized in Release 1 implementations.

5) Unless the database supports subtype checking, it is possible to UPDATE database values to values not belonging to the subtypes declared for their columns; requiring checking for this condition could have an unacceptable performance impact on an Ada/SQL system. For this reason, GR11 states that programs creating bogus data are erroneous. An implementation that can support database subtype checking may raise the DATA_EXCEPTION exception upon detecting a subtype constraint violation.

6) Release 1 implementations do not support the <update statement: positioned>.

8.12 <update statement: searched>

Function

Update rows of a table.

Format

<update statement: searched> ::=
UPDATE (<table name> ,
SET => <set clause: searched>
[{ and <set clause: searched> } ...] [,
WHERE => <search condition>]) ;

<set clause: searched> ::=
<object column: searched> <= { <value expression> | NULL_VALUE }

<object column: searched> ::= <column name>

Effective Ada Declarations

For a table t:

```
procedure UPDATE
          ( TABLE : in TABLE_NAME_t;
            SET   : in SET_CLAUSE_t;
            WHERE : in SEARCH_CONDITION := NULL_SEARCH_CONDITION );
```

See also all declarations, other than for the UPDATE procedure, given in 8.11.

Example

```
TERMINATED_EMPLOYEE : EMPLOYEE_NAME;
 . . .
UPDATE ( EMPLOYEE ,
SET   => SALARY  <= 0.0
     and MANAGER <= NULL_VALUE ,
WHERE => EQ ( NAME , TERMINATED_EMPLOYEE ) );
 . . .
UPDATE ( EMPLOYEE ,
SET   => SALARY <= 1.05 * SALARY ); -- cost of living raise
```

Syntax Rules

1) The applicable <privileges> for the <table name> shall include UPDATE for each <object column: searched>.

   NOTE: The "applicable <privileges>" for a <table name> are defined in 6.6, "<privilege definition>".

2) Let T denote the table identified by the <table name>. T shall not be a read-only table or a table that is identified in a <from clause> of any <subquery> that is contained in the <search condition>.

3) A <value expression> in a <set clause: searched> shall not include a <set function specification>.

4) Each <column name> specified as an <object column: searched> shall identify a column of T. The same <object column: searched> shall not appear more than once in an <update statement: searched>.

5) The scope of the <table name> is the entire <update statement: searched>.

6) For each <set clause: searched>:

   Case:

   a) If NULL_VALUE is specified, then the column designated by the <object column: searched> shall allow nulls.

   b) If NULL_VALUE is not specified, then the data type of the column designated by the <object column: searched> shall be the same as the data type of the <value expression>.

General Rules

1) Case:

   a) If a <search condition> is not specified, then all rows of T are the object rows.

   b) If a <search condition> is specified, then it is applied to each row of T with the <table name> bound to that row, and the object rows are those rows for which the result of the <search condition> is true. Each <subquery> in the <search condition> is effectively executed for each row of T and the results used in the application of the <search condition> to the given row of T. If any executed <subquery> contains an outer reference to a column of T, the reference is to the value of that column in the given row of T.

   NOTE: "Outer reference" is defined in 5.7, "<column specification>".

2) Each object row is updated as specified by each <set clause: searched>. A <set clause: searched> specifies an object column and an update value of that column. The object column is the column identified by the <object column: searched>. The update value is the null value or the value specified by the <value expression>. If the <value expression> contains a reference to a column of T, then the reference is to the value of that column in the object row before any value of the object row is updated. If there are no object rows to be updated, then the NO_DATA exception is raised.

3) An object row is updated in the following steps:

   a) A candidate row is created which is a copy of the object row.

   b) For each <set clause: searched>, the value of the specified object column in the candidate row is replaced by the specified update value.

   c) The object row is replaced by the candidate row.

4) If T is a viewed table defined by a <view definition> that specifies "WITH_CHECK_OPTION", and the <query specification> contained in the <view definition> specifies a <where clause>, then the <search condition> of that <where clause> shall be true for the candidate row; otherwise, the CONSTRAINT_VIOLATION exception is raised.

5) Let C denote an object column. Let v denote a nonnull update value of C.

6) Case:

   a) If C is of a character string data type and any character of v does not belong to the subtype of the characters in C, then the program executing the <update statement: searched> is erroneous.

   b) If C is of a character string data type and the length of v is equal to the maximum number of characters C can contain, then the value of C is set to v.

   c) If C is of a character string data type and can contain a maximum of L characters, and the length M of v is smaller than L, then the first M characters of C are set to v, and the last L-M characters of C are set to the space character.

   d) If C is of a character string data type and the length of v is greater than the maximum number of characters C can contain, then the program executing the <update statement: searched> is erroneous.

   e) If C is of an integer, floating point, or enumeration data type, and v belongs to the subtype of C, then the value of C is set to v.

   f) If C is of an integer, floating point, or enumeration data type, and v does not belong to the subtype of C, then the program executing the <update statement: searched> is erroneous.

Notes

1) Release 1 implementations do not support NULL_VALUE updates.

2) The Ada/SQL <update statement: searched> conforms to the ANSI SQL <update statement: searched>. The correspondence between Ada/SQL rules and ANSI SQL rules is as follows:

ANSI SQL	Ada/SQL	See Notes
SR1-SR5	SR1-SR5	-
SR6	SR6	3
GR1-GR3	GR1-GR3	-
GR4	GR4	4
GR5	GR5	-
GR6	GR6	5

3) SR6 expresses one aspect of Ada/SQL's strong typing. The length restriction of ANSI SQL SR6.*b is covered as* Ada/SQL GR6.d.

4) ANSI SQL GR4 qualifies the <where clause> of the <view definition> as "not contained in a <subquery>". A <subquery> cannot appear in a <view definition> to which GR4 applies, however. T must be an updatable table, and it is a consequence of other rules that a <subquery> cannot be used in the definition of an updatable table. The detection and/or reporting of check violations is not standardized in Release 1 implementations.

5) Unless the database supports subtype checking, it is possible to UPDATE database values to values not belonging to the subtypes declared for their columns; requiring checking for this condition could have an unacceptable performance impact on an Ada/SQL system. For this reason, GR6 states that programs creating bogus data are erroneous. *An implementation that can* support database subtype checking may raise the DATA_EXCEPTION exception upon detecting a subtype constraint violation.

9. Index

References are to section number. Bold type indicates defining reference; italics indicate reference in Example only.

- D -

Distribution List for IDA Memorandum Report M-362

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Capt Steve Myatt WIS JPMO/XPT Washington, D.C. 20330-6600	2 copies
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
Ms. Patricia L. Freeman MS-FP2/2255 TRW Command Support Division Federal Systems Group 12900 Fair Lakes Parkway Fairfax, VA 22033	2 copies
Mr. Fred Friedman P.O. Box 576 Annandale, VA 22314	1 copy
Ms. Patty Hicks DSAC-SR 3990 Broad St. Columbus, OH 43216-5002	1 copy
Ms. Elinor Koffee DSAC-SR 3990 Broad St. Columbus, OH 43216-5002	1 copy
Ms. Kerry Hilliard 7321 Franklin Road Annandale, VA 22003	1 copy
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 480 South Aiken Av. Pittsburgh, PA 15231	1 copy
Dr. C.E Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy
IDA	
General W.Y. Smith, HQ	1 copy
Mr. Philip Major, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Bill Brykczynski, CSED	50 copies
Ms. Audrey A. Hook, CSED	1 copy
Ms. Katydean Price, CSED	4 copies

NAME AND ADDRESS	NUMBER OF COPIES
IDA Control & Distribution Vault	3 copies